

510.84

I 26r

no.997-1005

1979-80

incompl.

copy 2







## CENTRAL CIRCULATION BOOKSTACKS

The person charging this material is responsible for its renewal or its return to the library from which it was borrowed on or before the **Latest Date** stamped below. **You may be charged a minimum fee of \$75.00 for each lost book.**

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

TO RENEW CALL TELEPHONE CENTER, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

When renewing by phone, write new due date below  
previous due date.

L162



Digitized by the Internet Archive  
in 2013

<http://archive.org/details/compilingfastpar1002spee>

510.84  
Il 6r  
no. 1002  
copy 2

UIUCDCS-R-80-1002

UIIU-ENG 80 1702  
COO-2383-0063

Q.

COMPILING FAST PARTIAL DERIVATIVES  
OF FUNCTIONS GIVEN BY ALGORITHMS

by

Bert Speelpenning



THE LIBRARY OF THE

MAY 29 1980

UNIVERSITY OF ILLINOIS  
URBANA-CHAMPAIGN

January 1980



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



UIUCDCS-R-80-1002

COMPILING FAST PARTIAL DERIVATIVES  
OF FUNCTIONS GIVEN BY ALGORITHMS

by

Bert Speelpenning

January 1980

DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN  
URBANA, ILLINOIS 61801

Supported in part by the U. S. Department of Energy, Grant US ENERGY/  
EY-76-S-02-2383 and submitted in partial fulfillment of the requirements  
of the Graduate College for the degree of Doctor of Philosophy.



## ACKNOWLEDGMENTS

It is a pleasure to acknowledge the support and guidance of my thesis supervisor, Professor C. William Gear. I am thankful for his easygoing style of letting me pursue my research essentially as I wanted. His research group has been an excellent place in which to work. The students in this group have been a steady source of stimulating discussions and friendship. The efforts of Al Whaley in maintaining, improving and explaining our local UNIX system have been greatly appreciated.

I also wish to thank the following professors for their guidance during earlier stages of my stay as a graduate student at Illinois: Professor Jurg Nievergelt (now at E.T.H. in Switzerland), Professor Thomas R. Wilcox (now at Intel Corporation) and Professor Arthur Sedgwick (now at Dalhousie University). They have helped to broaden my background considerably.

Barbara Armstrong has been of invaluable help in getting this thesis in readable form.

The research was supported by Department of Energy Contract US ENERGY /EY-76-S-02-2383.



## TABLE OF CONTENTS

	<u>Page</u>
1. INTRODUCTION.....	1
1.1. Object of this Research.....	3
1.2. Major Results of this Research.....	4
1.3. Outline of this Thesis.....	4
2. SYMBOLIC DIFFERENTIATION OF ALGORITHMS: PREVIOUS WORK.....	5
2.1. Warner, 1975.....	5
2.2. Joss, 1976.....	6
2.3. Kedem, 1977.....	8
2.4. Extensions to Gradients and Jacobians.....	9
2.5. Time and Space Requirements for the Algorithm Produced by Joss.....	10
2.6. Comparison of Joss with Numerical Differencing.....	10
3. ACCURACY CONSIDERATIONS.....	13
3.1. The Algorithm and the Function it Represents.....	14
3.2. Is Joss' Theorem Moot?.....	15
3.3. First Example: random search.....	16
3.4. Second Example: table lookup.....	17
3.5. Summary.....	18
4. COMPILATION OF EFFICIENT GRADIENTS.....	19
4.1. The Optimizing Compiler Approach to Improving Joss' Method: its limits.....	19
4.2. Compilation of Efficient Gradients: an outline.....	22
4.3. Joss' Method Viewed as a Sequence of Matrix Multiplications .....	22
4.4. On the Economics of Matrix Multiplication.....	26
4.5. The Problem of Factor Storage.....	29
4.6. An Interpretation of the Method not Based on Joss.....	31



5. COMPILATION OF FAST JACOBIANS.....	34
5.1. Finding Jacobians One Row at a Time.....	34
5.2. Comparison with Some Alternatives.....	36
5.3. Critical Analysis of One-Row-at-a-Time Jacobians.....	37
5.4. Optimal Multiplication of Factors for Obtaining a Jacobian.....	38
5.5. Extension to Arbitrary Flow of Control: run-time method.....	42
5.6. Extension to Arbitrary Flow of Control: compile-time method.....	44
6. IMPLEMENTATION.....	47
6.1. A User Description.....	47
6.2. How Jake Works.....	60
7. CONCLUSIONS.....	68
7.1. Experience with Jake.....	69
7.2. Future Work.....	70
7.3. Summary.....	73
REFERENCES.....	74
VITA.....	75



## 1. INTRODUCTION

There is a very peculiar situation within Numerical Analysis that has persisted for a long time. It concerns the problem of computing derivatives of functions conveniently, reliably and cheaply. One might have assumed that widely-used codes would exist that automate the process of taking derivatives of a function of given description yielding descriptions of the derivatives that execute fast when run on a computer. After all, the process of taking derivatives is well-understood mathematically, and the problem of computing derivatives is a recurrent one in many methods requiring functional iteration.

The reality is different. Several excellent numerical methods find very little use because they require knowledge of the partial derivatives of the function they are operating on. Numerical analysts in practice seem to avoid computing derivatives with a passion. Those who write down derivatives of complicated formulas by hand are almost universally driven to exasperation as the enterprise proves exceedingly error-prone. Where the computation of derivatives cannot be avoided, by far the most commonly used method is "numerical differencing." This method approximates a derivative by sampling the function in nearby points, computing the slope of a secant. By all conceivable standards, this method must be considered crude and primitive. The convergence rate of iteration schemes is often demonstrably lower if numerical differencing approximations are substituted for the true derivatives. Moreover, finding the gradient of a function  $f(x_1, \dots, x_n)$  through numerical differencing requires  $n+1$  function evaluations. This might seem reasonable, even unavoidable. But it is not. People who have had the tenacity to write down formulas for gradients by hand have observed that there is considerable redundancy between computations for the different components of the gradient. It has been postulated that the hand-coded computation of the gradient of a formula  $f$  need not cost more than a small number of function evaluations independent of  $n$ . Such observations might be of limited value because they apply only to cases where the function  $f$  is given by a rather simple closed form formula.

All the same it seems obvious that one must be able to improve substantially on numerical differencing. Numerical differencing regards the function as a black box, as a monolithic entity and hence is blind to the structure of the function. Surely, a method should be able to derive some advantage from being given access to the entire text of the function it is asked to differentiate.

How is it possible then that the situation outlined above has been allowed to persist for so long? It is not that symbolic methods for differentiation as such have been lacking; indeed, symbolic differentiation of expressions has been around almost as long as computers. Very sophisticated systems for symbolic algebraic manipulation exist, such as MACSYMA and FORMAC, and all offer facilities for differentiation of formulas as a matter of course. However, numerical analysts on the whole have not viewed these systems as adequate solutions to the problem. First, for most algebraic manipulation systems, a hand-translation is still required to get the output of these systems into a computer-executable form. Second, these systems are not geared to optimize entire gradient computations; instead they are geared to simplify individual derivatives; in other words, they are geared to satisfy the mathematician user, not the programmer user. Evaluation of the gradient will still take  $O(n)$  function evaluations. Third, and most importantly, algebraic manipulation systems deal with formulas in closed form, and are not set up to deal with functions given by arbitrary algorithms.

What is needed is a system as sketched in Figure 1.

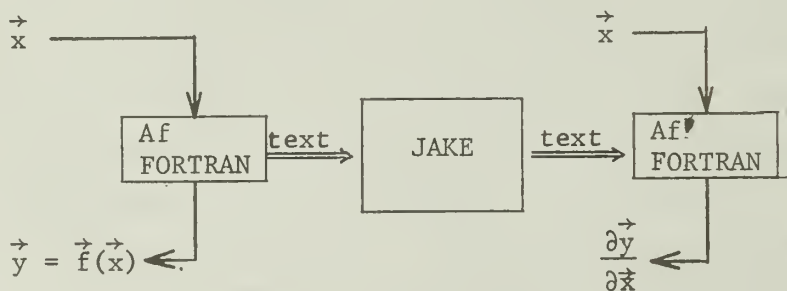


Figure 1. Use of Jake

Such a system will accept the text of an algorithm Af, embodying  $y=f(x)$

and written in a suitable programming language, and construct from it the text of an algorithm  $Af'$  that computes the Jacobian  $\frac{\partial y}{\partial x}$ . In fact, such systems do exist, though none produce algorithms  $Af'$  that can beat numerical differencing in terms of convenience, speed or memory requirements.

An explanation for the fact that no methods exist to automate the process of taking derivatives in a way that can successfully compete with numerical differencing may be found in the increasing specialization occurring within Computer Science. The separation of Numerical Analysis from Software is virtually complete and few people care to bridge the gap between the areas. In Numerical Analysis, the notion of programs that produce programs rather than numbers is largely absent. For most numerical analysts the FORTRAN compiler is completely transparent, as if Created on the same day as the computer. There is little awareness of language processing as a software writing tool in the sense of the products we have come to expect from places like Bell Labs. Notable exceptions include user languages for physical modeling and for statistical computations. Conversely, people involved in writing software tools may have a tendency to write only such software tools that aid in the writing of other software tools, and although this opens fascinating avenues of auto-catalysis, the real usefulness of these tools must ultimately come from application to outside areas. Tools are means to an end, not ends in themselves. What seems required is not merely cooperation between software people and numerical analysts but efforts by people with a certain minimal understanding and interest in both areas. The effort invested in such hetero-catalysis could pay off very handsomely.

### 1.1. Object of this Research

We set out to design and implement a system as in fig. 1 that would be general, convenient to use, and fast. Generality pertains to the class of algorithms  $Af$  it accepts. Convenience of use depends, among other things, on the number of changes the user needs to make in his algorithm  $Af$  before it is acceptable to the system. Speed pertains to

the algorithm  $Af'$  produced by the system:  $Af'$  should produce partial derivatives of  $f$  much faster than numerical differencing.

### 1.2. Major Results of this Research

A full solution to the problem of compiling fast gradients has been obtained. For the problem of compiling fast Jacobians of arbitrary shape a partial solution has been found. This thesis describes a method and its implementation capable of producing algorithms  $Af'$  that compute the gradient of a function  $f(x_1, \dots, x_n)$  in an amount of time equivalent to a constant number of function evaluations independent of  $n$ . The space requirements of the algorithm  $Af'$  are modest.

### 1.3. Outline of this Thesis

Chapters 2 and 3 deal with the feasibility of algorithmic differentiation. Sections 2.1 and 2.2 explain the method of Joss [JOS76], who showed in his Ph.D. thesis how one can assign a consistent and useful meaning to the notion of "derivative of an algorithm". Joss' method cannot compete with numerical differencing, as is shown in a detailed comparison of the performance of both methods in section 2.7. However, Joss' method is a convenient point of departure for a description of our own method.

The positive results of this thesis are detailed in Chapters 4, 5 and 6. Chapter 4 presents a new method for constructing fast gradients. Chapter 5 extends the method to the construction of fast Jacobians. Chapter 6 describes an actual compiler, Jake, implementing the theory presented. Conclusions, timing tests and suggestions for future work are presented in Chapter 7.

## 2. SYMBOLIC DIFFERENTIATION OF ALGORITHMS: PREVIOUS WORK

An algorithm  $A_f$  representing a mathematical function  $y=f(x)$  can be transformed by mechanical means into another algorithm  $A_{f'}$  that represents the derivative  $y' = \frac{\partial y}{\partial x}$ .

### 2.1. Warner, 1975

For very restricted algorithms  $A_f$  consisting merely of a sequence of assignment statements without any flow of control, this was noted and exploited by D.D. Warner in 1975 in a technical report from Bell Labs [WAR75]. His Partial Derivative Generator accepts a straight-line code program and compiles it into another that, when run, computes derivatives of the function represented by the original program. His generator rests on the use of the chain rule of differentiation. If the values of  $u$ ,  $\frac{\partial u}{\partial x}$ ,  $v$  and  $\frac{\partial v}{\partial x}$  are known (for a given value of  $x$ ), then

$$\frac{\partial (u*v)}{\partial x} = v * \frac{\partial u}{\partial x} + u * \frac{\partial v}{\partial x}$$

so that any assignment statement

$w := u * v$

in the original program may be replaced by

$dwdx := v * dudx + u * dvdx;$   
 $w := u * v;$

This applies, more generally, to any known operator  $op(u,v)$  such as "+", "-", "/", "max":

$w := u \text{ op } v$

is replaced by

$dwdx := \frac{\partial op}{\partial u} * dudx + \frac{\partial op}{\partial v} * dvdx;$   
 $w := u \text{ op } v$

where  $\frac{\partial op}{\partial u}$ ,  $\frac{\partial op}{\partial v}$  are known expressions in  $u$  and  $v$ .

For unary operators a similar result holds. Any straight-line computation can be easily broken down into unary and binary operations: that is how compilers compile expressions anyway. Trivial rules, such

as  $\frac{\partial \text{constant}}{\partial x} = 0$  and  $\frac{\partial x}{\partial x} = 1$  complete the picture and also constitute the inductive base in a proof of the correctness of Warner's method, a proof in which the chain rule provides the inductive step. But Warner doesn't really prove the correctness of his approach. It is apparently obvious to him that consistently replacing statements like

$$w := u \text{ op } v$$

by

$$dw dx := \frac{\partial \text{op}}{\partial u} * du dx + \frac{\partial \text{op}}{\partial v} * dv dx;$$

$$w := u \text{ op } v$$

in a program without flow of control leads to the correct computation of the derivative.

## 2.2. Joss, 1976

An important breakthrough was the doctor's thesis of Johann Joss at ETH in Switzerland in 1976 [JOS76]. Joss is concerned with algorithms (using Algol as a vehicle) that freely use if-then-else, goto and for-statements. The basic idea is again quite intuitive: for any given value of  $x$ , the program goes through a definite (and hopefully finite) sequence of assignment statements. That sequence of assignment statements, which might have been obtained from an execution trace, defines a straight-line program. For the particular value of  $x$ , the straight-line program would produce the same value for  $y$  as the original program. Moreover, we may reasonably expect that both programs produce the same value for  $y$  in some very small neighborhood of the point  $x$ . If this turns out to be true, we may differentiate the straight-line program. We know how to differentiate a straight-line program from Warner's work, and we know that his method does not radically change the structure of the original program; rather it is a mild expansion of it. All the original values are still being computed, and in the same sequence as in the original program. Viewing any flow of control in a program as a way to abbreviate the straight-line program (and also as a way to lay down several different straight-line programs in one single notation) suggests the following approach to symbolic differentiation of algorithms:

- 1) leave all flow-of-control statements untouched
- 2) leave all assignments to integer variables untouched
- 3) replace all statements assigning to a real variable by a pair of statements just as in straight-line programs.

For example:

```
A:      w := 0;
        for i := step 1 until n do
            w := w * x + a[i];
        y := exp(w);
```

will be replaced by (assuming array "a" contains constants):

```
B:      dwdx := 0;
        w := 0;
        for i := 1 step 1 until n do
            begin
                dwdx := dwdx * x + w;
                w := w * x + a[i]
            end;
        dydx := exp(w) * dwdx;
        y := exp(w);
```

for a given value of n, let's say  $n = 2$ , both A and B are equivalent to straight-line programs:

<pre>A ≡ w := 0;     i := 1;      w := w * x + a[i];     i := 2;      w := w * x + a[i];      y := exp(w);</pre>	<pre>B ≡ dwdx := 0; w := 0;     i := 1;     dwdx := dwdx * x + w;     w := w * x + a[i];     i := 2;     dwdx := dwdx * x + w;     w := w * x + a[i];     dydx := exp(w) * dwdx;     y := exp(w);</pre>
--	---

Here we see that Warner's method indeed underlies Joss' method, and the same applies for other values of  $n$ .

In case the straight-line equivalent of a program not only depends on the value of some unknown parameter  $n$  but also on the particular value of  $x$ , a more sophisticated approach is needed to show the validity of the method. Joss proves in his thesis that the method outlined does indeed produce the correct results under quite general conditions. The most limiting condition in practice is the condition that computer arithmetic be exact. Real variables are assumed to hold real values of infinite precision. Chapter 3, on numerical accuracy, discusses the seriousness of the exact arithmetic assumption. What the assumption allows Joss to prove is that the transformed algorithm computes the correct derivative "for almost all" values of  $x$ . More precisely, there can only be countably many real values of  $x$  for which the derivative comes out wrong. Most of these values  $x$  are on the dividing line created by an if, as the value 0 in

$$y := \text{if } x < 0 \text{ then } -x \text{ else } x .$$

Such values very often correspond to points where the derivative  $\partial y / \partial x$  does not exist in the first place, so strange answers in such points are generally wholly acceptable.

Joss' thesis is remarkable for its clarity and brevity. The fact that it was written in German may have restricted its wider dissemination.

### 2.3. Kedem, 1977

Gershon Kedem published a paper in the proceedings of the 1977 U.S. Army Numerical Analysis and Computer Conference outlining ideas very similar to those of Joss yet not developed as far [KED77]. Kedem's paper appears to be the first publication in English showing the feasibility of symbolic differentiation of full-fledged programs. The implementation described by Kedem is not particularly impressive, and it is obvious that Kedem was not aware of Joss' thesis. Deserving praise for independently discovering differentiation of algorithms, and still providing the only English source of its description, Kedem nevertheless

is not the originator of the idea. Joss came first.

#### 2.4. Extensions to Gradients and Jacobians

It is immediately obvious that methods to produce derivatives of a scalar function of a scalar variable can be extended to produce gradients and Jacobians. The gradient of a function  $y = f(x_1, \dots, x_n)$  is the row vector  $\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}$ . The Jacobian of a set of functions

$$\begin{aligned} y_1 &= f_1(x_1 \dots x_n) \\ &\vdots \\ y_m &= f_m(x_1 \dots x_n) \end{aligned}$$

is the  $m * n$  matrix  $J$  with  $J_{ij} = \frac{\partial y_i}{\partial x_j}$ .

The extension of symbolic differentiation to a general function  $\vec{y} = \vec{f}(\vec{x})$  described by a subroutine  $F(X,Y,N,M)$  with  $X,Y$  vectors of arbitrary size  $N,M$  is important because most applications deal with functions of many variables.

Warner, Joss and Kedem all considered such extensions: Warner, Joss and Kedem are all able to produce gradients; Warner and Joss also produce Jacobians; Kedem is able to produce first, second and higher order derivatives.

Warner's system, though able to produce Jacobians, is not as powerful as it may sound: all subscripts in array references are restricted to constants so in essence they behave as ordinary scalars.

Kedem and Joss allow true array indexing (computable subscripts) and hence need the additional differentiation rule:

$$\frac{\partial x[i]}{\partial x[j]} = \text{"if } i = j \text{ then } 1 \text{ else } 0\text{"}$$

No theoretical problems arise from  $x$  being a vector. Joss does not mention gradients in the theoretical part of his thesis at all; gradients suddenly enter the description of his implementation. Instead of pairing each real variable  $u$  in the original program with a new scalar  $dudx$  representing the value  $\frac{\partial u}{\partial x}$ , he pairs each variable  $u$  with an array  $dudx[1:n]$  whose elements  $dudx[j]$  represent the current value of  $\frac{\partial u}{\partial x_j}$ . Instead of replacing " $w := u * v$ " by

$$dwdx := v * dudx + u * dvdx;$$

$w := u * v$

he replaces it by:

```

begin
  for j := 1 step 1 until n do
    begin
      dwdx[j] := v * dudx[j] + u * dvdx[j];
    end;
  w := u * v
end

```

Clearly, this works. For Joss, who seems primarily interested in giving a feasibility proof where no feasibility was known previously, such an approach is sufficient. Whether the approach is optimal is not immediately clear and this issue deserves investigation.

## 2.5. Time and Space Requirements for the Algorithm Produced by Joss

If the original algorithm  $Af(\vec{x}, \vec{y})$ , for a certain value of  $\vec{x}$ , takes  $T$  time to run to completion with space requirements  $S$ , then the program  $Af'(\vec{x}, \vec{y}, J)$  produced by Joss to compute the Jacobian  $J = \partial \vec{y} / \partial \vec{x}$  will run to completion in  $O(nT)$  time and require  $O(nS)$  space, where  $n$  is the size of the vector of independent variables  $\vec{x}$ .

## 2.6. Comparison of Joss with Numerical Differencing

As mentioned in Chapter 1, numerical differencing is a widely-used alternative to symbolic differentiation of algorithms. Numerical differencing is based on sampling the original function in the neighborhood of the point  $\vec{x}$  and therefore does not even need to see the text of the algorithm  $Af(\vec{x}, \vec{y})$ , it merely needs to call it.

In comparing numerical differencing with Joss' method, the following criteria are relevant:

- a) ease
- b) numerical accuracy

- c) time requirements
- d) space requirements

The comparison will be made for the computation of the gradient

$$\frac{\partial y}{\partial x_1} \dots \frac{\partial y}{\partial x_n} \text{ from}$$

```

SUBROUTINE F(X,N,Y)
REAL X(N),Y
.
.
.
END

```

#### 2.6.1. Comparison: ease

There can be no doubt that numerical differencing is easier:

```

SUBROUTINE GRADF(X,N,Y,GRAD)
REAL X(N),Y,GRAD(N)
DATA DELTA/ ... /
CALL F(X,N,Y)
DO 10 I = 1,N
    X(I) = X(I) + DELTA
    CALL F(X,N,YNEW)
    GRAD(I) = (YNEW - Y) / DELTA
    X(I) = X(I) - DELTA
10  CONTINUE
RETURN
END

```

Except for the complication of choosing DELTA, this is basically all there is to numerical differencing. Joss' method, or any form of symbolic differentiation, cannot compete with that.

### 2.6.2. Comparison: accuracy

Chapter 3 is devoted to issues of accuracy. We will anticipate here our main conclusion: in the presence of round-off it is very difficult to predict whether symbolic differentiation will give more accurate answers than numerical differencing (with DELTA chosen optimally) for any given algorithm Af.

### 2.6.3. Comparison: time

If the original subroutine  $F(X,N,Y)$  requires  $T$  time for a particular value of  $X$ , Joss requires  $O(nT)$  for the gradient, and so does numerical differencing.

### 2.6.4. Comparison: space

Numerical differencing requires extra space only in the form of the gradient itself:  $S + n$ . Joss requires  $O(nS)$ , as all real scalars and real arrays are accompanied by arrays and matrices to hold the  $n$  derivatives with respect to  $\vec{x}$  of all values computed.

### 3. ACCURACY CONSIDERATIONS

The thrust of this thesis is to present a method of producing symbolic derivatives that represents an improvement over previous methods in terms of speed and of space requirements. In developing the new method (described in the following chapters of this thesis), no explicit consideration was given to issues of accuracy in the presence of round-off. After the method was developed, it was easily seen that in terms of accuracy it mirrors the method of Joss from chapter 2 in many relevant aspects. Joss, in his thesis, touches on accuracy considerations for his method, but mostly by implication.

It is outside the scope of this thesis to develop a theory of the numerical behavior of symbolic derivatives under round-off. This is not intended to convey the impression that numerical behavior under round-off is somehow not important. Numerical behavior is one of several factors that affect the user's confidence in the answers produced by a certain method. Fortunately, it is possible to address the issue of user confidence in a meaningful way even without having a theory of round-off. Other problems than round-off are associated with symbolic differentiation and they may well be the bigger problem at this point in the development and acceptance of symbolic methods.

To be used, a program must produce answers in which the user can have some confidence. In the scientific and cultural climate of today, people are quite ready, initially, to accept answers from a computer program, but a small number of unpleasant surprises with the program will turn the same people sharply against that program. One does not make a method stronger by hiding its weaknesses.

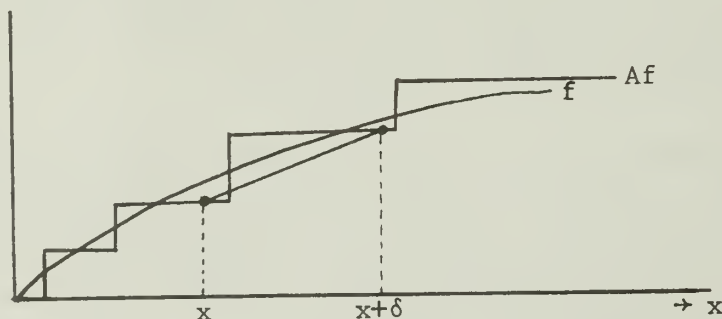
Symbolic differentiation of algorithms does have some pitfalls in the sense that it can be misapplied to produce outrageous results. As these pitfalls can be avoided rather easily provided one is aware of their existence, it is important to point out where these pitfalls lie. These pitfalls affect accuracy of the results in a more dramatic way even than round-off and therefore this issue belongs in this chapter.

### 3.1. The Algorithm and the Function it Represents

An algorithm  $Af$  ready to be differentiated does not arise in a vacuum. Rather, the algorithm was written to represent or approximate some mathematical function  $f$ . The algorithm is secondary to the mathematical function and there may be discrepancies between the two for a variety of technical reasons. For one, the mathematical function  $f$  may be known only implicitly, e.g. as obeying a functional equation  $G(f)=0$ . Such a function can often be represented by an algorithm only by use of iteration. Second, the mathematical function  $f$  may be known imperfectly, e.g. only on a subinterval or by its values in certain points. The algorithm  $Af$  may be using some interpolation technique to provide an approximation to  $f$  on the entire interval of interest.

A key assumption of symbolic differentiation of algorithms is that not only  $Af$  approximates  $f$ , but  $(Af)'$  approximates  $f'$  as well.

Symbolic differentiation uses the text of the algorithm  $Af$  as its sole source of knowledge about the function  $f$ . So the best one can hope to achieve with symbolic differentiation is to obtain the exact derivative  $(Af)'$  of  $Af$ . In what respect is this different from any other computer method such as numerical differencing? At first glance it would appear that numerical differencing has an even bigger handicap, as it merely samples the algorithm  $Af$  at some points but is never allowed even to inspect its text. Yet the following graph suggests that the situation is not nearly as simple as that:



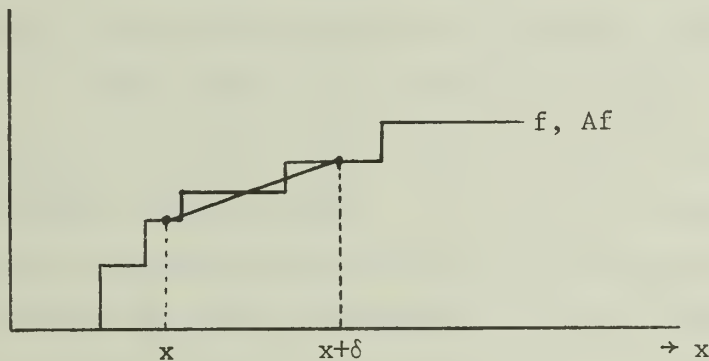
If  $Af$  approximates  $f$  through a step function,  $(Af)'$  will be zero ("almost everywhere") no matter how closely  $Af$  approximates  $f$ . In

contrast,

$$\frac{Af(x+\delta) - Af(x)}{\delta}$$

may be fairly close to  $f'$  provided  $\delta$  is large relative to the step size. A large  $\delta$  places a lower bound on the truncation error and hence will not give a very accurate result, but most likely the numerical derivative will be in the proper range, whereas symbolic differentiation misses the true derivative of  $f$  altogether.

The prevalence of approximations that achieve quite small uniform error bounds and that do so by "tacking" seems to confer a rather "unfair" advantage to numerical differencing. Such advantage for numerical differencing in terms of accuracy is by no means universal, however. It is easy to give examples where numerical differencing is way off the mark whereas symbolic differentiation is exact, e.g.:



In this example,  $Af$  and the numerical derivative are the same as before. But now the step in the function  $Af$  is not merely a technical artifact, the step is there because the function  $f$  itself happens to have just such a step.

### 3.2. Is Joss' Theorem Moot?

Joss' main theorem deals with an algorithm  $Af$ , its symbolic derivative  $(Af)'$  and a real interval  $I$  for  $x$  on which both algorithms are being considered. Under the assumption of infinite precision arithmetic, the algorithm  $Af$  defines a function  $F$ . In contrast to the previous section we are not considering the function  $f$  that  $Af$  was intended to approximate, we are now considering the function  $F$  that is

defined by  $Af$  as is. Instead of somehow bounding the error in  $(Af)'$  seen as an approximation of  $f'$ , can we obtain at least a statement about the accuracy of  $(Af)'$  seen as an approximation to  $F'$  ?

Joss' result is that indeed such a statement can be made, and in particular, that the following can be proved:

$$F' = (Af)' \quad \text{"almost everywhere" on the interval } I.$$

The phrase "almost everywhere" means that there are at most countably many points where the equality does not obtain. Another formulation of the same result is that  $F'$  and  $(Af)'$  are equal in the  $L_2$  norm.

The problem with Joss' theorem is not that its proof is incorrect or faulty but rather that it is not applicable to any real machine. Real machines do not meet the requirement of infinite precision arithmetic. The damaging fact here is not so much the presence of round-off in real arithmetic per se, but rather that the representation of real values in the memory of the machines must needs be finite. In the interval  $I$  for  $x$  on the real axis, only a finite number of points are representable within the machine. Those values comprise a set  $M$  of measure zero. So Joss' theorem allows that symbolic differentiation produces values that are correct (i.e. equal to those of  $F'$ ) for all  $x$  in  $I$  except for those values of  $x$  that are representable in the machine! In other words, Joss' theorem proves nothing about the accuracy of the values of  $(Af)'$  evaluated (even without round-off) in the points  $x$  of  $M$ , and it is only these values that are accessible at all and hence of any practical significance.

The next two sections show example algorithms where indeed strange (probably even counterintuitive) results are obtained. On closer analysis, however, neither result provides a counterexample to Joss' theorem and the algorithms are very contrived indeed. The examples given are intended as a warning.

### 3.3. First Example: random search

In the procedure below, `random()` is assumed to be a perfect pseudo-random generator, producing numbers between 0 and 1.

```

procedure slowid(x,y);
begin comment 0<x<1;
L:   y:=random();
      if( x≠y ) goto L;
end

```

It is not claimed that the procedure slowid is a practical way of computing the function  $y=x$  : it may be rather slow. In fact, it is not obvious that slowid will converge for all  $x$  in the interval from 0 to 1. Convergence for all  $x$  would place a very stringent burden on the random generator. Yet if convergence is only to be guaranteed for values of  $x$  representable in the machine, it is not hard to show how the random generator could be written to make slowid converge for all representable  $x$ . Any process that would cycle through all representable values in the interval would do instead of the random generator.

In any event, it is not entirely unreasonable to say that slowid represents the function  $y=x$ . At the same time,  $(\text{slowid})' = 0$ , not 1, for all  $x$ .

#### 3.4. Second Example: table lookup

Assume that floating point numbers of the machine are represented by a word of  $w$  bits. Assume that the memory of the machine has more than  $2^w$  words. The first  $2^w$  words of memory can be used as a table, implementing any conceivable function  $y=F(x)$  as follows:

- 1) take  $x$ , examine its bit pattern ( $w$  bits).
- 2) use the bit pattern as an address into the memory.
- 3) retrieve a word of  $w$  bits from the memory at that address.
- 4) return the retrieved word as the result  $y$ .

The code for steps 1-4 is the algorithm Af and can be thought of as stored above the table in the memory.

Symbolic differentiation again produces zero for every floating point number  $x$ , regardless of what function was stored in the table.

### 3.5. Summary

A reader for whom the results obtained for the functions in the previous sections seem intuitively incorrect is advised not to use symbolic differentiation of algorithms. However, it is not particularly hard to change one's perspective such that the results obtained for those functions become intuitively correct. This is perhaps more a reflection on intuition than on symbolic differentiation as such.

## 4. COMPILATION OF EFFICIENT GRADIENTS

For reasons that will become clear soon, this chapter focuses on gradients rather than Jacobians of general shape and size. Chapter 5 will generalize the results of this chapter to compilation of efficient Jacobians.

In Chapter 1 it was argued that symbolic differentiation of algorithms as developed by Joss could not compete with numerical differencing and Chapter 2 showed this in more detail. To make symbolic differentiation competitive one must improve significantly on the programs produced by Joss' method. One approach which appears very promising is to replace Joss' compiler, which is a one-pass non-optimizing compiler, by an optimizing compiler incorporating all the latest program optimization techniques and more. In the early stages of the research leading to this thesis, much time was devoted to pursuing the optimizing compiler approach and it was found to be not ultimately successful. The next section will outline this approach and suggest why an optimizing compiler staying within the framework of the "Joss interface" should not be expected to effect significant speed-ups for a significant class of algorithms. After that, we will turn to positive results.

#### 4.1. The Optimizing Compiler Approach to Improving Joss' Method: its limits

First we show a example algorithm A that allows speedup by a factor of  $O(n)$  over Joss.

<p>A: <math>y := 1;</math></p> <p style="padding-left: 2em;"><u>for</u> <math>i := 1</math> <u>step</u> 1 <u>until</u> <math>n</math> <u>do</u></p> <p style="padding-left: 4em;"><math>y := y * x[i];</math></p>	<p>B: <math>\vec{g} := \vec{0}; y := 1;</math></p> <p style="padding-left: 2em;"><u>for</u> <math>i := 1</math> <u>step</u> 1 <u>until</u> <math>n</math> <u>do</u></p> <p style="padding-left: 4em;"> <math display="block">\left\{ \begin{array}{l} \vec{g} := \vec{g} * x[i] \\ \quad \quad \quad + y * \text{unit}(i); \\ y := y * x[i]; \end{array} \right.</math> </p>
---	--

<pre> C:  y := 1;     for i := 1 step 1 until n do       y := y * x[i];     for i := 1 step 1 until n do       g[i] := y/x[i]; </pre>	<pre> D:  y := 1;     for i := 1 step 1 until n do       { lp[i] := y;         y := y * x[i];       }     rp := 1;     for i := n step -1 until 1 do       { g[i] := lp[i] * rp;         rp := rp * x[i];       } </pre>
---	--

Algorithm A computes  $y = \prod_{i=1}^n x_i$ . Algorithm B computes  $\vec{g} = \partial y / \partial \vec{x}$  according to Joss in  $O(n^2)$  time. The vector notation in algorithm B abbreviates a loop over the  $n$  components of the vector. Algorithm C is the first indication that an  $O(n)$  algorithm might be found for  $\vec{g}$ . It is clear that  $\frac{\partial y}{\partial x_i} = \prod_{j \neq i} x_j$  so it is tempting to try  $\frac{\partial y}{\partial x_i} = \frac{y}{x_i}$ . However, algorithm C will fail if any  $x_i$  is zero. Algorithm D avoids any division and still realizes the  $O(n)$  time bound of algorithm C. Algorithm D is based on the identity

$$\frac{\partial y}{\partial x_i} = l_i * r_i \text{ where } l_i = \prod_{j < i} x_j \text{ and } r_i = \prod_{j > i} x_j.$$

A question worth considering is whether D could have been obtained from B by an automatic method. Indeed, B can be transformed into D by steps leaving the semantics of the algorithm invariant. The steps can be constrained to be those in the Irvine Catalogue [STA76], for instance. However, the path of algorithms and transformations between B and D is a very tortuous one, and it is very hard to see how an automatic procedure would find that path even if it "knew" that it was supposed to end up with algorithm D. Certainly, one would not want to rely on theorem proving techniques for any real-life size problem. It should also be remarked that D cannot be obtained from B by merely exploiting sparsity in  $\vec{g}$  or in  $\text{unit}[i]$ . That only serves to bring down the operation count from  $n^2$  to  $n^2/2$ . Rather, D involves a rearrangement of the entire loop structure. The next question is whether perhaps D could have been obtained from A directly, by some automatic method. Or, rather, whether such a method can be general enough to handle a large class of program structures of interest. Here one thinks of a repertory

of special techniques for special loop structures, extended with a set of transformations that map more general loops into those special structures. Typically, one has no hope of recognizing entire program structures, but one may have the hope that it proves sufficient to focus merely on innermost loops, the rationale being that speeding up innermost loops by  $O(n)$  will speed up the entire program by the same order of magnitude. This approach has been used quite successfully by Kuck and collaborators [KUC78] in the context of optimizing programs for execution on parallel machines.

A theory for so-called "scalar recurrences" was developed as a generalization of algorithm A, but there is no justification for believing that local optimization such as optimization of innermost loops will achieve much for general algorithms. Consider algorithm F below. It computes  $y = \det(X)$  where  $X$  is a square matrix of size  $N$ . Algorithm F itself requires  $O(N^3)$ , while computing the gradient  $\partial y / \partial X[i,j]$  for  $i = 1, \dots, N$ ,  $j = 1, \dots, N$  according to Joss requires  $O(N^5)$ . Yet optimizing this by hand will give a method that is  $O(N^3)$ , hence we save a factor  $O(N^2)$ . As " $n$ " equals  $N^2$  here, the savings are  $O(n)$  as before. Nothing like an optimization of the innermost loop of F would accomplish such savings. If any doubt as to this point remains, we can add partial pivoting to F, and see how that destroys any possibility of giving a closed form characterization of what the inner loop does, let alone of how to optimize it.

```
F:  y := 1;
    for i := 1 step 1 until N do
      {
        y := y * X[i,i];
        for j := i+1 step 1 until N do
          {
            mult := X[j,i]/X[i,i];
            for k := j step 1 until N do
              X[j,k] := X[j,k] - mult * X[i,k];
          }
      }
```

To optimize  $\partial y / \partial X$  by hand, we may use the easily derived formula

$$\frac{\partial \det(X)}{\partial X} = \det(X) \cdot X^{-1}$$

Both  $\det(X)$  and  $X^{-1}$  can be computed in  $O(N^3)$  and multiplied in  $O(N^2)$ . This results in an  $O(N^3)$  algorithm. Note that algorithm A can be regarded as a special case of algorithm F with  $X$  a diagonal matrix.

#### 4.2. Compilation of Efficient Gradients: an outline

We now turn to the major positive results of this thesis. It is indeed possible to achieve a speed-up of  $O(n)$  for gradients over Joss' approach with a method not intrinsically more complicated than his. To show this we have to view his method from a much greater height and with far less concern for local efficiency than the optimizing compiler of the last section did. We have to abstract and focus on what all the examples of the last section and indeed all programs from which to produce gradients have in common, and that is that they take the information contained in  $n$  numbers  $x[1], \dots, x[n]$  and from it produce a single scalar value  $y$ . We will show that Joss' method is algebraically equivalent to a sequence of matrix multiplications, the last one of which does not involve a square matrix but a row vector. It is easy to see that if those matrices were full, multiplying those same matrices in a different sequence would lead to identical results but  $O(n)$  faster. It will be shown that the extreme sparsity of the matrices (i.e., the matrices have mostly zero entries) can be exploited in a meaningful way, and the resulting method is  $O(n)$  faster than Joss without making excessive demands on memory space.

#### 4.3. Joss' Method Viewed as a Sequence of Matrix Multiplications

The theory will be presented using an example algorithm. As usual,  $\vec{x}$  represents the independent variables,  $y$  is the dependent variable, and  $\partial y / \partial \vec{x}$  is to be constructed.

```

for i := 1 step 1 until n do
  begin
    t := 0;
    for j := 1 step 1 until n do
      t := t + a[i,j] * x[j];
    z[j] := t;
  end

```

```

y := 1;
for i := 1 step 1 until n do
    y := y * z[i];

```

The method of Joss attempts to keep the matrix J up to date at all times:

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial x_1} & \dots & \frac{\partial x_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial x_n}{\partial x_1} & & \frac{\partial x_n}{\partial x_n} \\ \frac{\partial t}{\partial x_1} & & \frac{\partial t}{\partial x_n} \\ \frac{\partial z_1}{\partial x_1} & & \frac{\partial z_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial z_n}{\partial x_1} & & \frac{\partial z_n}{\partial x_n} \\ \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

The variables making up the rows of J:  $\vec{x}$ ,  $t$ ,  $\vec{z}$ ,  $y$  comprise the "state space"  $S$ . A point in the state space characterizes the values of all real variables in the program. Any time during execution of the program, the state of the memory is given by a point in the state space, and J will contain its Jacobian. When an assignment statement such as

$y := y * z[i]$  is executed, the point in the state space moves, and the matrix J must be updated. Joss generates the extra statements

```

for jg := 1 step 1 until n do
    yg[jg] := yg[jg] * z[i] + y * zg[i,jg];

```

This can be abbreviated as:

$$\vec{y_g} := \vec{y_g} * z[i] + y * \vec{z_g[i]};$$

Due to the chain rule, the right hand side of this statement is linear in  $\vec{y_g}$  and  $\vec{z_g[i]}$ , and therefore it is possible to describe the update to

J as a matrix product:

$${}^{\text{"y"}} \begin{bmatrix} \quad \quad \quad \end{bmatrix}_{J'} := {}^{\text{"y"}} \begin{bmatrix} \text{"z[i]"} & \text{"y"} \\ 1 & \circ \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & 1 \\ \circ & y \\ & z[i] \end{bmatrix}_F * \begin{bmatrix} \text{"z[i]"} \\ \text{"y"} \end{bmatrix}_J$$

The matrix F, called a "factor", is determined by the partial derivatives of the original expression  $y * z[i]$ , which partial derivatives are placed on the row corresponding to the left hand side "y" in the state space, in the columns corresponding to the right hand sides, "z[i]" and "y", in the state space.

It is important to observe that the factor F is a Jacobian matrix in its own right. F is the Jacobian of the transformation  $S \rightarrow S$  that moves points in the state space according to  $y := y * z[i]$ . We can see this readily by introducing some abbreviations and consistently using accent marks to distinguish new values from old values. J is abbreviated as

$$\begin{array}{c} \backslash \partial \vec{x} \\ \partial \vec{x} \\ \partial t \\ \partial \vec{z} \\ \partial y \end{array} \begin{bmatrix} \quad \quad \quad \end{bmatrix}, \text{ or even as } \partial(\vec{x}, t, \vec{z}, y) / \partial(\vec{x})$$

Similarly,  $J'$  is abbreviated as

$$\begin{array}{c} \backslash \partial \vec{x}' \\ \partial \vec{x}' \\ \partial t' \\ \partial \vec{z}' \\ \partial y' \end{array} \begin{bmatrix} \quad \quad \quad \end{bmatrix}, \text{ or as } \partial(\vec{x}', t', \vec{z}', y') / \partial(\vec{x})$$

Now F is seen to be

$$\begin{array}{c} \backslash \partial \vec{x} \quad \partial t \quad \partial \vec{z} \quad \partial y \\ \partial \vec{x}' \\ \partial t' \\ \partial \vec{z}' \\ \partial y' \end{array} \begin{bmatrix} \quad \quad \quad \end{bmatrix}, \text{ or } \partial(\vec{x}', t', \vec{z}', y') / \partial(\vec{x}, t, \vec{z}, y)$$

The "deep truth" underlying the statement  $J' := F * J$  now shows up as

$$\partial(\vec{x}', t', \vec{z}', y') / \partial(\vec{x}) := \partial(\vec{x}', t', \vec{z}', y') / \partial(\vec{x}, t, \vec{z}, y) * \partial(\vec{x}, t, \vec{z}, y) / \partial(\vec{x})$$

So far, we have looked at a single assignment statement, and the effect it has on  $J$ . Now we must look at the effect of a whole sequence of assignments on  $J$ . From Joss' work we know that it is sufficient to consider straight-line code, the straight-line code being thought of as deriving from an execution trace.

It should be clear that

$$J_{\text{final}} = F_s * F_{s-1} * \dots * F_1 * J_{\text{initial}},$$

where the factors  $F_1, \dots, F_s$  have been indexed by the order in which the assignment statements that gave rise to them were executed.

In the equation given above,  $J_{\text{initial}}$  is simply:

$$\begin{array}{c} \backslash \partial \vec{x} \\ \partial \vec{x} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\ \partial t \\ \partial \vec{z} \\ \partial y \end{array}$$

The final result is not  $J_{\text{final}}$ , but a single row extracted from it:

$$\partial(y) / \partial(\vec{x}) = \vec{g} * J_{\text{final}}, \text{ where } \vec{g} = (0 \ 0 \ 0 \ 0 \ \dots \ 0 \ 1).$$

Summing up we have

$$\partial(y) / \partial(\vec{x}) = \vec{g} * F_s * F_{s-1} * \dots * F_2 * F_1 * J_{\text{initial}}$$

In Joss' method, this matrix product is evaluated from right to left. Though the factors  $F_1, \dots, F_s$  must of necessity become available in that order, it is not necessary that they be used in the same order. Matrix multiplication is associative, and it may turn out that a different order of multiplication is faster. Indeed, we will show in the next section that multiplication from left to right is an order of magnitude faster:

$$\partial(y) / \partial(\vec{x}) := (((\dots(\vec{g} * F_s) * F_{s-1}) \dots) * F_1) * J_{\text{initial}};$$

Any order of multiplication different from the order of generation of the factors raises the issue of storage space for the factors. For every execution of any assignment statement a factor becomes available (not merely once for its presence in the program text), so at first glance the storage problem appears truly overwhelming and this would seem to rule out any change in the multiplication order. However, it will be shown how the factor storage problem can be solved very neatly, resulting in a method that typically requires far less memory space than Joss' approach.

#### 4.4. On the Economics of Matrix Multiplication

We showed that Joss' method is algebraically identical to a sequence of matrix multiplications

$$\vec{g} * F_s * \dots * F_1 * J_{\text{initial}}$$

evaluated from right to left, where both  $\vec{g}$  and  $J_{\text{initial}}$  are merely slices of the unit matrix. All factors  $F_1$  are square,  $m * m$ , where  $m$  is the dimension of the state space. The matrix  $\vec{g}$  is really a row vector,  $1 * m$ ; the matrix  $J_{\text{initial}}$  is  $m * m$ .

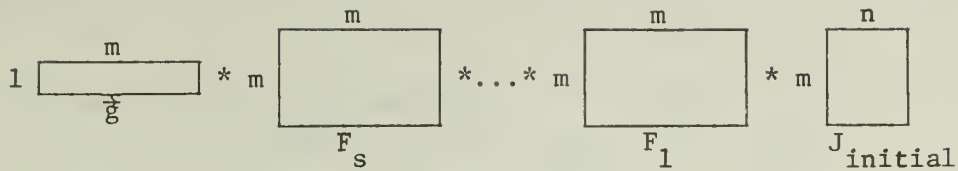
This section will show that evaluating the product from left to right is  $O(n)$  faster. First, the two ways of evaluating the product will be compared using the assumption that all matrices  $F_1$  are full. We know, of course, that the  $F_1$  matrices are not full at all; we must then verify that the comparison still holds true given the special structure the factors possess. The reason we bother making the comparison for full matrices at all is its great heuristic value.

Multiplying a full  $p * q$  matrix  $A$  with a full  $q * r$  matrix  $B$  results in a  $p * r$  matrix  $C$  in  $p * q * r$  operations. This assumes that the standard algorithm is used:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Clever algorithms for multiplying square matrices such as Strassen's [STR69] are left undiscussed if only because we cannot subsequently generalize the results to sparse systems.

The matrix product



if evaluated from the right, has intermediary results all of size  $m * m$  and hence costs

$$s(m^2n) \text{ operations,}$$

ignoring the last "multiplication" with  $g$ . Evaluating the same product from left to right, all intermediary results have size  $1 * m$  and hence we chalk up

$$s(m^2) \text{ operations,}$$

this time ignoring the last "multiplication" with  $J_{\text{initial}}$ . Clearly, left-to-right is  $O(n)$  faster.

Now we turn to the analysis of the left-to-right multiplication exploiting the very special structure of the factors  $F$ .

Consider the product

$$g_0 = g * F_s * \dots * F_1$$

evaluated as

$$g_s := g$$

$$g_{i-1} := g_i * F_i \text{ and focus on a particular product}$$

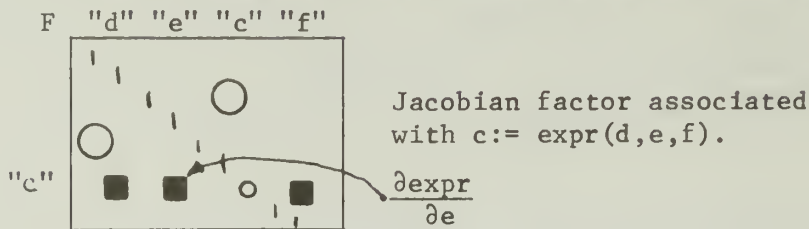
$$g := g' * F$$

where  $F$  is a factor associated with an assignment statement. Take as a typical assignment statement the following:

$$c := \text{expr}(d, e, f);$$

Here, "expr" is any expression using  $+$ ,  $-$ ,  $*$ ,  $/$  and standard functions; the left hand variable  $c$  and the right hand variables  $d$ ,  $e$ ,  $f$  may actually be subscripted. "Expr" may involve additional variables, e.g. integer variables, as long as they are not in the state space. What does  $F$  look like? It differs from the unit matrix in at most four

places. These are all in the row corresponding to  $c$ . The column positions of those changes are those corresponding to  $c$ ,  $d$ ,  $e$  and  $f$ .



Now  $g = g' * F$  can be computed as follows:

```

for i := 1 step 1 until n do
    g[i] := g'[i];
t := g["c"];
if t ≠ 0 then
    begin
        g["c"] := 0;
        g["d"] := g["d"] + t * F["c", "d"];
        g["e"] := g["e"] + t * F["c", "e"];
        g["f"] := g["f"] + t * F["c", "f"];
    end
end

```

Here, "c", etc., has been used as a convenient notation for the row number corresponding to variable  $c$  in the state space.

A further simplification is possible if  $\vec{g}$  and  $g^*$  are made to share the same space in memory. Surely, once  $\vec{g}$  is computed,  $g^*$  is no longer of interest. Surely, too, the above algorithm allows  $\vec{g}$  and  $g^*$  to share space; indeed, it becomes faster.

$\vec{g} := \vec{g} * F$  can be computed as follows:

```

t := g["c"];
if t ≠ 0 then begin g["c"] := 0;
    g["d"] := g["d"] + t * F["c", "d"];
    g["e"] := g["e"] + t * F["c", "e"];
    g["f"] := g["f"] + t * F["c", "f"];
end;

```

We have now arrived at the conclusion that the time required to multiply a row vector with a factor matrix is independent of  $n$ . The

time required is easily bounded by a constant times the time required for evaluating the right hand side from which the factor was derived. A bound of the same form, again independent of  $n$ , can easily be obtained for the time required to obtain the few nonidentity elements of the factor matrix at the time the original statement is executed.

These bounds should be contrasted with Joss' own method, which entails expanding every assignment

```
c := expr(d,e,f)
```

into an array assignment requiring  $O(n)$  operations:

```
pcpd := ∂expr/∂d;
pcpe := ∂expr/∂e;
pcpf := ∂expr/∂f;
for jg := 1 step 1 until n do
    cg[jg] := pcpd * dg[jg] + pcpe * eg[jg]
            + pcpf * fg[jg];
c := expr(d,e,f);
```

#### 4.5. The Problem of Factor Storage

If the proposed technique of carrying out the multiplications from left to right is to be viable, a solution must be found to the problem of fast storage and retrieval of factors without requiring an excessive amount of memory to do so.

First, it should be remarked that the factors have very few entries worth storing explicitly. The factor associated with

```
c := expr(d,e,f)
```

can be easily reconstructed (but never needs to be; what counts is that it is determined by it) from the three values

$$\frac{\partial \text{expr}}{\partial d}, \frac{\partial \text{expr}}{\partial e}, \frac{\partial \text{expr}}{\partial f}$$

plus the values of four integers giving the row corresponding to  $c$ , and the columns corresponding to  $d, e, f$ . In fact,

$$\partial c \begin{array}{|c|c|c|} \hline \partial d & \partial e & \partial f \\ \hline \end{array} : F$$

is a convenient notation for the factor  $F$ , as the next section will

show.

Even after each factor is seen to involve only a small number of items to be stored, it is still not possible to give an a priori bound on the amount of storage required: the number of factors  $s$  depends on the number of assignments (involving variables in the state space) that is actually executed. In Joss' method, storage requirements are given by  $m * n$ ; in the new method storage requirements can only be bounded as a constant times the running time  $T$  of the program. The essential difference between the two situations is that Joss' storage must be random-access whereas large portions of the factor storage may be on secondary store such as disk or "backspace-able" tape. The way the new method accesses the factors is strictly like a stack: last in-first out, with no factors going out before all factors are in. Hence a block of central memory can be set aside as a buffer. Factors are put into the buffer when they are generated; if the buffer threatens to overflow, it is written out to disk. Conversely, on multiplying the factors, they are read from the buffer; if the buffer becomes empty, the previous one is read in from disk.

With this organization, the number of disk accesses  $D$  is related to the total storage requirement  $S$  (linked linearly to the running time  $T$  of the original program) and the buffer size  $B$  as follows:

$$D = 2S/B \quad (\text{disregarding rounding})$$

As the cost of disk access is largely dependent on  $D$  and only very weakly dependent on  $S$ , it should always be possible to minimize disk activity by increasing the buffer size. Yet essentially there is no minimum buffer size and so the minimum memory requirements of the method are very low. There is full flexibility for achieving a suitable trade-off between disk activity and memory use. In any event, the costs are proportional to  $T$ , not  $nT$  as in Joss. It should be stressed that in addition to the buffer of size  $B$ , the method only requires room for the  $\vec{g}$  vector, size  $m$ . It will be recalled that  $m$  was the dimension of the state space and therefore bounded by the memory size required by the original program. To sum up, the method requires about twice as much space as the original program plus whatever you can spare for a buffer,

with the obvious trade-off between buffer size and disk activity. In contrast, Joss uses  $(n+1)$  times the space of the original program.

#### 4.6. An Interpretation of the Method not Based on Joss

Useful as it was to derive the new method from Joss' method because it obviates proving that flow of control can be ignored to the extent that it was ignored, it is also very convenient and instructive to have an interpretation of the method based directly on the chain rule. This interpretation involves the row vector  $\vec{g}$ . The value of  $g["v"]$  for some variable  $v$ , is the "current" value of

$$\partial y / \partial v.$$

Consider the straight-line code corresponding to an execution trace of the program and focus on the last part of it. As an example, let the last three statements be:

```
s-2: w := 2 * z[j];
s-1: u := v + z[i] * w;
s:   y := u * v;
```

We introduce the notation " $=\theta$ " as in

$$y =\theta u * v$$

to mean that the final value of  $y$  is a function of the values of the variables of the right hand side  $u, v$ , as they were just before statement  $s$  was executed.

The function of  $y =\theta u * v$  has a gradient:

$$\partial y / \partial \vec{x} =\theta fsu * \partial u / \partial \vec{x} + fsv * \partial v / \partial \vec{x}$$

As " $\partial / \partial \vec{x}$ " is common to all terms, it may be left implied giving

$$\partial y =\theta fsu * \partial u + fsv * \partial v$$

Here  $fsu$ ,  $fsv$  are numbers from the factor matrix  $F_s$ , given by:

$$fsu =\theta v$$

$$fsv =\theta u$$

Statement  $s-1$  is similarly characterized by

$$u =\theta \textcircled{s-1} v + z[i] * w;$$

$$\partial u =\theta \textcircled{s-1} fs-lv * \partial v + fs-lz[i] * \partial z[i] + fs-lw * \partial w$$

Consider now the joint effect of statements s-1, s. Together they define y as a function of u, v, z[i], w as follows:

$$y = \textcircled{s-1} u * v = \textcircled{s-1} (v + z[i] * w) * v$$

This process of characterizing the semantics of assignment statements by appropriate substitution is well-known [DIJ78]. A similar process of substitution gives:

$$\partial y = \textcircled{s-1} f_{su} * \partial u + f_{sv} * \partial v$$

$$= \textcircled{s-1} f_{su} * (f_{s-lv} * \partial v + f_{s-lz[i]} * \partial z[i] + f_{s-lw} * \partial w) + f_{sv} * \partial v$$

$$= \textcircled{s-1} (f_{su} * f_{s-lv} + f_{sv}) * \partial v + f_{su} * f_{s-lz[i]} * \partial z[i] + (f_{sv} * f_{s-lw}) * \partial w$$

For the discerning eye, these substitution steps are seen to be identical to the way the  $\vec{g}$  vector changed when post multiplied by a factor matrix! The factor matrix  $F_s$  associated with statements can be conveniently abbreviated as:

$$\partial y \begin{array}{|c|c|} \hline \partial u & \partial v \\ \hline f_{su} & f_{sv} \\ \hline \end{array}$$

The previous section mentioned this abbreviation in the context of a concern for efficient storage of the factor. Now we see that the notation is more than that.

We can define multiplication of two objects

$$\partial y \begin{array}{|c|c|} \hline \partial u & \partial v \\ \hline f_{su} & f_{sv} \\ \hline \end{array} \quad \text{and} \quad \partial u \begin{array}{|c|c|c|} \hline \partial v & \partial z[i] & \partial w \\ \hline f_{s-lv} & f_{s-lz[i]} & f_{s-lw} \\ \hline \end{array}$$

in two ways: one is by expanding both to full factor matrices, multiplying and contracting again, and the second is by our newly interpreted process of direct substitution:

$$\begin{aligned} \partial y \begin{array}{|c|c|} \hline \partial u & \partial v \\ \hline f_{su} & f_{sv} \\ \hline \end{array} * \partial u \begin{array}{|c|c|c|} \hline \partial v & \partial z[i] & \partial w \\ \hline f_{s-lv} & f_{s-lz[i]} & f_{s-lw} \\ \hline \end{array} &= \partial y \begin{array}{|c|c|c|c|} \hline \partial w & \partial z[i] & \partial w & \partial v \\ \hline f_{sn}^* & f_{sn}^* & f_{sn}^* & f_{sv} \\ \hline f_{s-lv} & f_{s-lz[i]} & f_{s-lw} & \\ \hline \end{array} \\ &= \partial y \begin{array}{|c|c|c|} \hline \partial v & \partial z[i] & \partial w \\ \hline f_{sv}+f_{sn}^* & f_{sn}^* & f_{sn}^* \\ \hline f_{s-lv} & f_{s-lz[i]} & f_{s-lw} \\ \hline \end{array} \end{aligned}$$

To see whether the procedure has been really understood, it is helpful to go one step further in backward direction to incorporate statement s-2 into the description, too. Statement s-2 is characterized by

$$\partial w \begin{array}{|c|} \hline \partial z[j] \\ \hline f_{s-2z[j]} \\ \hline \end{array}$$

So we have

$$\frac{\partial}{\partial y} \begin{matrix} \backslash \\ \partial v \end{matrix} \begin{matrix} \partial z[i] \\ a \end{matrix} \begin{matrix} \partial w \\ b \end{matrix} \begin{matrix} \\ c \end{matrix} * \frac{\partial}{\partial w} \begin{matrix} \backslash \\ \partial z[j] \end{matrix} \begin{matrix} fs-2z[j] \end{matrix} = \frac{\partial}{\partial y} \begin{matrix} \backslash \\ \partial v \end{matrix} \begin{matrix} z[i] \\ a \end{matrix} \begin{matrix} \partial z[j] \\ b \end{matrix} \begin{matrix} c*fs-2z[j] \end{matrix}$$

The question may now arise as to what happens in case  $i$  and  $j$  have equal values. The answer is: the procedure still works correctly. In the gradient notation it is quite acceptable to have something like

$$\frac{\partial}{\partial y} \begin{matrix} \backslash \\ \partial z[i] \end{matrix} \begin{matrix} \partial z[i] \\ p \end{matrix} \begin{matrix} \\ q \end{matrix}$$

It would in all respects be identical to

$$\frac{\partial}{\partial y} \begin{matrix} \backslash \\ \partial z[i] \end{matrix} \begin{matrix} p + q \end{matrix}$$

The latter form saves some space and some arithmetic, but both are correct, and hence in cases one doesn't know whether two expressions  $z[i]$  and  $z[j]$  refer to the same variable, one assumes simply that they don't. The algorithm to multiply the  $\vec{g}$  vector with a factor

$$\frac{\partial}{\partial c} \begin{matrix} \backslash \\ \partial d \end{matrix} \begin{matrix} \partial e \\ fcd \end{matrix} \begin{matrix} \partial f \\ fce \end{matrix} \begin{matrix} \\ fcf \end{matrix}$$

as given previously and as adapted below, deals correctly with all permutations of possible identity between  $c, d, e, f$ :

```
t := g[∂c];
if(t≠0) then begin
    g[∂c] := 0;
    g[∂d] := g[∂d] + t * fcd;
    g[∂e] := g[∂e] + t * fce;
    g[∂f] := g[∂f] + t * fcf;
end;
```

## 5. COMPILATION OF FAST JACOBIANS

In Chapter 4 it was shown how the gradient of the function  $y = f(x_1, \dots, x_n)$  can be constructed in a time proportional to  $T$ , where  $T$  is the time required for evaluating the function  $f$  itself. This chapter will attempt to generalize the results of Chapter 4 to find the Jacobian of a system of functions

$$y_1 = f_1(x_1, \dots, x_n)$$

.

.

.

$$y_k = f_k(x_1, \dots, x_n)$$

given as an algorithm  $Af(\vec{x}, \vec{y})$ . Let  $T$  again denote the time required for execution of  $Af$  and let  $S$  be the amount of memory involved. The method of Joss computes the Jacobian  $J = \partial y_i / \partial x_j$  in  $O(nT)$  time and  $O(nS)$  space. The next section shows that a straightforward extension of the method of Chapter 4 can compute Jacobians in  $O(kT)$  time and  $O(S)$  space (not counting space for the final Jacobian itself). So even the straightforward extension is superior to Joss in terms of space requirements. A comparison of these methods with regard to time will depend on  $k$  and  $n$ . If  $k \ll n$ , Joss' method loses out; if  $k > n$ , Joss' method is superior. In practice, however, the case  $k > n$  is very rare and may safely be ignored. If  $k \approx n$ , a comparison is more difficult and depends on the overhead associated with either method. In particular, for the important case  $k = n$ , Joss' method will usually be faster, but at most by a small factor independent of  $n$ .

### 5.1. Finding Jacobians One Row at a Time

Jacobians consist of rows, each row being a gradient. Row  $i$  is the gradient of  $y_i = f_i(x_1, \dots, x_n)$ . For a given  $i$ , therefore, we could choose to regard  $y_i$  as the output variable and compute its gradient. Or, to stay closer to the formalism developed in Chapter 4, we could

precede the procedure exit of Af with the statement

$z := y[1];$

and regard  $z$  as the output variable. What it boils down to is that

$$\left\{ \frac{\partial y_1}{\partial x_j}, j = 1, \dots, n \right\} = \vec{g}_1 * F_s * \dots * F_1 * J_{\text{initial}}$$

where  $\vec{g}_1$  is a unit vector in the state space with the one in the position corresponding to  $y_1$ .

In passing, we point out that if one's real object is to compute  $\vec{\alpha}J$  for some row vector  $\vec{\alpha}$ ; one never need construct the Jacobian  $J$  explicitly; instead one may compute

$$\vec{\alpha} \cdot J = \vec{g}_{\vec{\alpha}} * F_s * \dots * F_1 * J_{\text{initial}}$$

where  $\vec{g}_{\vec{\alpha}}$  is a vector with values  $\alpha_1$  on the positions corresponding to the  $y_1$ , and with zeros everywhere else. This corresponds to finding the gradient of  $z$  with these statements inserted just before exit of Af:

```

z := 0;
for i := 1 step 1 until k do
  z := z + α[i] * y[i];

```

Computing the full  $k * m$  Jacobian of the vector  $\vec{y}$  would comprise the following steps:

- 1) do the computation of  $\vec{y}$  according to the Af, and emit factors along the way;
- 2) for each  $i$ ,  $1 \leq i \leq k$ , do:
  - a) set the  $\vec{g}$  vector to the unit vector corresponding to  $y_1$ ;
  - b) multiply factors into  $\vec{g}$ :
 
$$\vec{g} := \vec{g} * F_j, \quad j = s, s-1, \dots, 1;$$
  - c) extract the gradient of  $y_1$  from  $\vec{g}$ . This produces the  $i$ -th row of the Jacobian.

This description shows clearly that the method requires  $O(kT)$  time; it also shows the time required to be far less than  $k$  times as much as that for a single gradient: factor emission need not be repeated.

## 5.2. Comparison with Some Alternatives

The following is an alternative to the one-row-at-a-time approach from the previous section:

- 1) Emit all factors;
- 2) Initialize the  $k * m$  matrix  $g$  so that the  $i$ -th row of  $g$  is the unit vector corresponding to  $y_i$ ,
- 3) Multiply  $g$ , from the right, by factors:
 
$$g := g * F_j, j = s, s-1, \dots, 1;$$
- 4) Extract the columns of  $g$  that belong in the Jacobian and throw away the rest.

This variant also requires  $O(kT)$ , though this is not as easy to see as in the previous section. The space requirements, however, have gone up to  $O(kS)$ . What we gain is that we need not read in the factors more than once. Still, a memory requirement of  $O(kS)$  seems an exceptionally severe penalty to pay. For large  $k, S$  the space will simply not be available, whereas for smaller  $k$  a more powerful and flexible way to keep down the overhead in reading factors is to increase the size of the factor space buffer.

The main reason we introduced the variant above is that it is the most direct right-to-left counterpart of Joss' method. Joss also requires a lot of storage:  $O(nS)$ . The close correspondence between the method of the previous section and its variant in this section suggests that Joss' method can be easily modified to reduce storage requirements to  $O(S)$ . In fact, this can be done:

Joss (modified):

- 1) For each variable  $u$ , allocate a new variable  $dudx_j$ ;

2) For each  $x_j$ ,  $1 \leq j \leq n$ , do:

- a) initialize all  $dudx_j$  to zero, except for  $dx[j]x_j$ , which is to be set to one;
- b) add to each assignment statement  $v := \text{expr}(u_1, u_2)$  the statement

$$dvdx_j := \frac{\partial \text{expr}}{\partial u_1} * du_1 dx_j + \frac{\partial \text{expr}}{\partial u_2} * du_2 dx_j;$$

- c) put all variables  $dy[l]dx_j \dots dy[k]dx_j$  in the  $j$ -th column of the Jacobian.

There is no need to compute  $\frac{\partial \text{expr}}{\partial u_1}$ , etc., each time: they could be emitted to a factor space (organized here as a queue) and read repeatedly. This allows for the same space-time tradeoffs by manipulating the factor storage buffer size.

This modification of Joss is so straightforward and so obviously advantageous in terms of storage that it is perhaps surprising that Joss never mentions it in his thesis. It constitutes strong evidence that Joss was only interested in feasibility of differentiation of algorithms, not its cost.

### 5.3. Critical Analysis of One-Row-at-a-Time Jacobians

The method of producing Jacobians described in section 5.1 is general, convenient and economical on space while being reasonably fast. For  $k \ll n$ , the method may well be perfectly adequate; clearly, it is for  $k = 1$ . The implementation described in Chapter 6 does, in fact, employ this method.

In the remainder of Chapter 5 several ideas will be introduced that may eventually lead to a compiler producing Jacobians that run significantly faster than those of Joss for all  $k \leq n$ . As these ideas are necessarily more tentative than those described in Chapter 4, the treatment is less detailed. Certainly one can skip reading the remainder of Chapter 5 and proceed to Chapter 6 without loss of continuity.

The reason that the one-row-at-a-time approach is not necessarily optimal is quite simple. The method for finding gradients in Chapter 4 was successful precisely because it exploited the fact that the  $\vec{g}$  vector in

$$\partial y / \partial \vec{x} = \vec{g} * F_s * F_{s-1} \dots * F_1 * J_{\text{initial}}$$

had only one row, so that left-to-right multiplication is  $O(n)$  faster. As soon as a Jacobian  $\partial \vec{y} / \partial \vec{x}$  is desired, for  $\vec{y}$  a vector of more than one element,  $\vec{g}$  becomes a matrix, and for  $k \approx n$ , the matrices  $g$  and  $J_{\text{initial}}$  will have similar shape, removing the advantage of one multiplication direction over the other. Yet there is no a priori reason to assume that only pure left-to-right multiplication or pure right-to-left multiplication can be performed. Associativity of matrix multiplication allows many other multiplication orders. Perhaps it is possible to find an optimal or near-optimal order of multiplication that is compatible with the results of Chapter 4 in the special case that  $g$  is a row vector. In the next section we will explore optimal multiplication of factors and the problems associated with it.

#### 5.4. Optimal Multiplication of Factors for Obtaining a Jacobian

In this section we will look at various multiplication orders for the product

$$g * F_s * F_{s-1} * \dots * F_1 * J_{\text{initial}}$$

for a given string of factors  $F_1, \dots, F_s$ .

To find the optimal order for one such particular product of factors should prove interesting even if there is at present no guarantee that such an optimal order could be found for an algorithm with arbitrary flow of control, as the algorithm encompasses a variety of straight-line programs each emitting strings of factors differing in value, number and configuration from one another.

Each factor  $F$  is a Jacobian matrix in its own right (cf. section 4.3) and so is a product of consecutive factors. Every Jacobian factor has an out-set  $F_{\text{out}}$  and an in-set  $F_{\text{in}}$  corresponding to the (single)

output variable and (any number of) input variables of the assignment statement from which  $F$  originated. So the  $F_{out}$  and  $F_{in}$  of the factor  $F$  emitted for

$$f := f * u + t$$

would be  $\{f\}$  and  $\{f, u, t\}$ , respectively. The factor matrix  $F$  will be identity, except for the "f" row which is nonzero only in the "f", "u" and "t" columns. The concepts  $F_{out}$  and  $F_{in}$  generalize to arbitrary products of factors. The Jacobian  $F$  of a mapping from  $F_{in}$  to  $F_{out}$  will be an identity matrix ( $m * m$ ), except for the rows corresponding to  $F_{out}$  which can be nonzero only in the columns corresponding to  $F_{in}$ . The nontrivial entries of  $F$  are therefore those on the intersection of a row from  $F_{out}$  and a column from  $F_{in}$ . To simplify the analysis we will regard all of these nontrivial entries as potentially nonzero and hence ignore any finer structure a factor may possess. This way, without actually performing the matrix multiplications we can keep track of the resulting  $F_{out}$  and  $F_{in}$  sets and express operation counts in terms of these.

Before we carry this out, we note that the concepts  $F_{out}$  and  $F_{in}$  apply to products of factors, not necessarily to products involving  $g$  or  $J_{initial}$ . Multiplication by  $g$  or  $J_{initial}$  serves essentially to throw out matrix terms that are now known to be irrelevant. We know from Chapter 4 how crucial it is to anticipate what is going to be thrown out eventually so as to avoid computing it in the first place. In a product

$$(g * F_4 * F_3) * (F_2 * F_1 * J_{initial})$$

all effort that goes into computing a certain row of  $(F_2 * F_1 * J_{initial})$  is wasted if the corresponding column of  $(g * F_4 * F_3)$  is all zero. Therefore, we introduce the concepts  $Y_{needed}$  and  $X_{dependent}$  as follows:

$Y_{needed}^{(i)}$  is the set of variables that correspond to a nonzero column in

$$g * F_s * \dots * F_{i+1}$$

$X_{dependent}^{(i)}$  is the set of variables that correspond to a nonzero row in

$$F_{i-1} * F_{i-2} * \dots * F_1 * J_{initial}$$

Any factor  $F_1$  with out-set  $F_{out}^{(1)}$  and in-set  $F_{in}^{(1)}$  that does not satisfy  $F_{out}^{(1)} \subseteq Y_{needed}^{(1)}$  and  $F_{in}^{(1)} \subseteq X_{dependent}^{(1)}$  may be simplified by throwing away terms that are apparently ultimately irrelevant. The effect is to set

$$F_{out}^{(1)} := F_{out}^{(1)} \cap Y_{needed}^{(1)}$$

$$F_{in}^{(1)} := F_{in}^{(1)} \cap X_{dependent}^{(1)}$$

and a similar simplification may be made everywhere along the way.

We are now ready to present recurrence formulas for  $F_{out}^{(1..j)}$  and  $F_{in}^{(1..j)}$  of the product.

$$F_{1..j} = F_1 * F_{1-1} * \dots * F_j$$

as well as for  $Y_{needed}$  and  $X_{dependent}$ .

$$Y_{needed}^{(s)} = \{y[1], \dots, y[k]\}$$

$$Y_{needed}^{(1)} = Y_{needed}^{(i+1)}, \text{ if } Y_{needed}^{(i+1)} \cap F_{out}^{(1)} = \emptyset$$

$$= (Y_{needed}^{(i+1)} \setminus F_{out}^{(1)}) \cup F_{in}^{(1)}, \text{ otherwise.}$$

$$X_{dependent}^{(1)} = \{x[1], \dots, x[n]\}$$

$$X_{dependent}^{(1)} = X_{dependent}^{(i-1)} \setminus F_{out}^{(1)}, \text{ if } X_{dependent}^{(i-1)} \cap F_{in}^{(1)} = \emptyset$$

$$= X_{dependent}^{(i-1)} \cup F_{out}^{(1)}, \text{ otherwise.}$$

$$F_{out}^{(1..1)} = F_{out}^{(1)}$$

$$F_{in}^{(1..1)} = F_{in}^{(1)}$$

$$F_{out}^{(1..j)} = (F_{out}^{(1..j+1)} \cup F_{out}^{(j)}) \cap Y_{needed}^{(1)}$$

$$F_{in}^{(1..j)} = F_{in}^{(1..j+1)} \cap X_{dependent}^{(j)}, \text{ if } F_{out}^{(j)} \cap F_{in}^{(1..j+1)} = \emptyset$$

$$= ((F_{in}^{(1..j+1)} \setminus F_{out}^{(j)}) \cup F_{in}^{(1)}) \cap X_{dependent}^{(j)}, \text{ otherwise.}$$

An estimate for the operation count for a product

$$F_{1..j} * F_{j-1..p}$$

$$|F_{out}^{(1..j)}| * |F_{in}^{(1..j)}| \quad |F_{out}^{(j-1..p)}| * |F_{in}^{(j-1..p)}|$$

where  $|\cdot|$  denotes the number of variables in a set.

The optimization problem has now been formulated in a manner that can be attacked by dynamic programming [BEL]. Unfortunately, dynamic programming requires time  $O(s^3)$  whereas multiplication in any order is never worse than  $O(nT)$ , where of course  $T=O(s)$ . So it seems evident that we must lower our goals and recognize that a heuristic approach to finding an approximately optimal solution is called for.

An example may serve to make the problem more palatable. The algorithm:

```

for i := 1 step 1 until k do
begin w := ln(x[i]);
  for j := 1 step 1 until n do
    w := w + a[i,j] * x[j];
  y[i] := w;
end;

```

when run with  $n = 3$  and  $k = 2$  will lead to the following sequence of factors:

$$\begin{aligned}
 &g * y_2 \begin{array}{c} w \\ \boxed{\phantom{000}} \\ 10 \end{array} ; w \begin{array}{c} w x_3 \\ \boxed{\phantom{000}} \\ 9 \end{array} ; w \begin{array}{c} w x_2 \\ \boxed{\phantom{000}} \\ 8 \end{array} ; w \begin{array}{c} w x_1 \\ \boxed{\phantom{000}} \\ 7 \end{array} ; w \begin{array}{c} x_2 \\ \boxed{\phantom{000}} \\ 6 \end{array} ; y_1 \begin{array}{c} w \\ \boxed{\phantom{000}} \\ 5 \end{array} ; w \begin{array}{c} w x_3 \\ \boxed{\phantom{000}} \\ 4 \end{array} ; \\
 &w \begin{array}{c} w x_2 \\ \boxed{\phantom{000}} \\ 3 \end{array} ; w \begin{array}{c} w x_1 \\ \boxed{\phantom{000}} \\ 2 \end{array} ; w \begin{array}{c} x_1 \\ \boxed{\phantom{000}} \\ 1 \end{array} * J_{initial}
 \end{aligned}$$

where  $g = \text{diag}(0,0,0,0,1,1)$  and  $J_{initial} = \text{diag}(1,1,1,0,0,0)$  if the state space is ordered as  $(x_1, x_2, x_3, w, y_1, y_2)$ .

$$\text{So } F_{4..2} = w \begin{array}{c} w x_3 x_2 x_1 \\ \boxed{\phantom{00000}} \end{array} ; F_{7..4} = w \begin{array}{c} w x_3 x_2 \\ \boxed{\phantom{000}} \\ y_1 \end{array}$$

In this example, the optimal ordering is easily seen to be:

$$F_{10..6} = (\dots(F_{10} * F_9) \dots F_6),$$

$$F_{5..1} = (\dots(F_5 * F_4) * \dots * F_1),$$

$$\partial(\vec{y})/\partial(\vec{x}) = g * F_{10..6} * F_{5..1} * J_{\text{initial}}.$$

The logic behind this is the same as we encountered in Chapter 4. Left-to-right multiplication is advantageous for a product of factors that has a single-element  $F_{\text{out}}$ . A string of consecutive factors such as  $F_{10}, \dots, F_6$  that has a product with a single-element  $F_{\text{out}}$  will be called a funnel. Replacing all funnels by their products will result in a substantial reduction over the one-row-at-a-time approach. When all funnels have been replaced by their products, we might then search for long strings with a product having a two-element  $F_{\text{out}}$ , and so on. However, in anticipation of the implementation issues of the next sections we should remark that there is a certain cost associated with changing multiplication sequence midstream. The savings due to multiplying funnels first are the most sweeping; it is the most easily recognized and the most easily implemented. The remaining multiplications can then probably be performed best with the one-row-at-a-time scheme.

#### 5.5. Extension to Arbitrary Flow of Control: run-time method

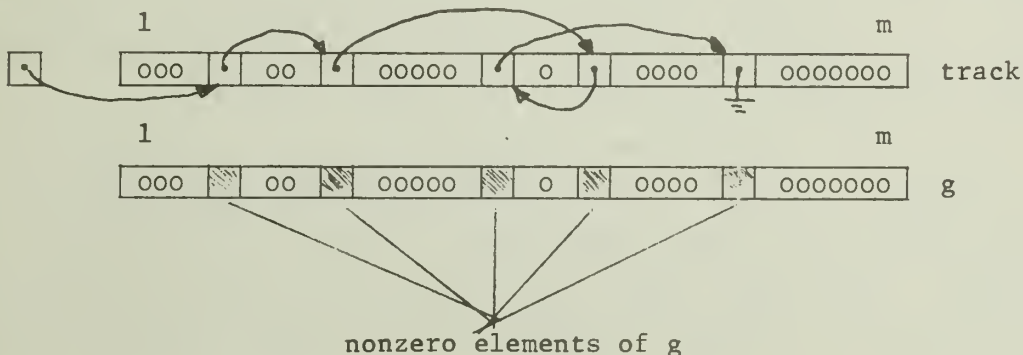
The approach outlined in the previous section can be extended to algorithms with arbitrary flow of control in two essentially different ways.

The first way is to delay the determination of multiplication order until run-time, after all factors have already been emitted; this will be the subject of this section.

The second way is to determine the multiplication order at compile time using flow graph analysis techniques; that will be the subject of the next section.

When the multiplication order is not being determined until all factors have been emitted, the situation is in every detail as sketched in the previous section. Implementation of the funnel multiplication involves setting up a separate factor storage space organized as a first-in, first-out queue. This new factor storage space will house

original factors and products of funnels. Multiplication of the factors in the new space can be done by any conceivable method such as one-row-at-a-time. In order to be able to recognize funnels we will need an array of flags of size  $m$  (the dimension of the state space). Initially, only the flags for  $y[1] \dots y[k]$  are set. Without loss of generality we may assume that the last factor emitted has an  $F_{out}$  consisting of the single element  $y[i]$ . We now start the row vector  $\vec{g}$  of size  $m$  as a unit vector corresponding to  $y[i]$ , we turn off the  $y[i]$ -flag and read in factors as if to compute the gradient of  $y[i]$ . But before multiplying a factor  $w$   $\begin{smallmatrix} u & v \\ \hline \end{smallmatrix}$  into  $\vec{g}$ , we check the  $w$ -flag. If the  $w$ -flag is set it will mean that another gradient computation (e.g. the gradients of  $y[1]$ ) is interested in the gradient of  $w$  and has been suspended to await the computation of the gradient of  $w$ . So on finding the  $w$ -flag set, the algorithm would locate all nonzero terms of  $\vec{g}$ , set the flag for all nonzero entries found in  $\vec{g}$ , shape the nonzero entries into a new factor and emit it to the new factor storage space. Then it would clear the  $w$ -flag and proceed to read and multiply factors into  $\vec{g}$  (reinitialized now to the unit vector corresponding to  $w$ ), again until a factor is read in with an output variable whose flag is set. This process is repeated until the entire factor storage space has been read and all intermediary products stored in the new factor storage space. It will be clear that compression and reinitialization of the row vector  $\vec{g}$  will require  $O(m)$  operations unless special precautions are taken. A data structure that is a hybrid between a row vector and a linked list structure may be required to keep track of all the nonzero positions of  $g$ . A sketch of such a data structure is presented below but without additional commentary.



### 5.6. Extension to Arbitrary Flow of Control: compile-time method

The main advantage of trying to determine a multiplication order at compile-time is that various forms of overhead associated with determining the multiplication order at run-time do not occur. As a result, the Jacobian computation for the special case  $k=1$  need not be slower than a gradient computation performed according to Chapter 4. The disadvantage is that flow graph analysis lacks the degree of resolution that can be achieved with a method having an entire execution trace available. Hence certain strings of factors will not be recognized as funnels even if they are. Flow graph analysis assumes worst case behavior. For example, it cannot distinguish between  $u[i]$  and  $u[j]$  because it knows nothing about subscripts; it will assume that a certain derivative  $\partial v / \partial x_1$  is nonzero as long as there is any path at all to the point under consideration on which path a value is assigned to  $v$  that can be traced back to  $x$ , whether or not that path will ever actually be executed. The concepts  $Y_{\text{needed}}$  and  $X_{\text{dependent}}$  can be approximated using flow graph analysis; they are now sets associated with points in the program, not sets associated with factors directly. The relationship between the two is that the set  $Y_{\text{needed}}$  associated with a certain point in the program will contain as subsets all the sets  $Y_{\text{needed}}$  associated with those factors that are emitted whenever flow of control reaches that point in the program. The  $F_{\text{out}}, F_{\text{in}}$  sets can also be approximated by flow graph analysis. For single assignment statements

```
s1:  u := v * w;
s2:  a[i] := b * w;
```

we have

$$F_{\text{out}}^{s1} = \{u\}; \quad F_{\text{in}}^{s1} = \{v, w\}; \quad F_{\text{out}}^{s2} = \{a\}; \quad F_{\text{in}}^{s2} = \{a, b, w\}.$$

We will list a set of conditions that guarantees that a certain block of code represents a funnel. We also claim that flow graph analysis can find such blocks, but neither claim will be proved. The conditions are as follows. The block must have a single exit, though it may have multiple entries. The last statement of the block should be an

assignment statement, e.g.

s:  $p := g * r$ ;

such that

a) the left hand variable  $p$  is a scalar;

b)  $p$  is  $Y_{\text{needed}}^{(s)}$ .

Let  $L$  be the set  $Y_{\text{needed}}^{(s)} \setminus \{p\}$ . None of the statements in the block may have a lefthand variable that is an element of  $L$ .

The power, as well as the limitations, of this type of flow graph analysis can be illustrated by contrasting two algorithms:

A:     for  $i := 1$  step 1 until  $k$  do  
           begin  $y[i] := 0$ ;  
               for  $j := 1$  step 1 until  $n$  do  
                    $y[i] := y[i] + a[i,j] * x[j]$ ;  
           end;

B:     for  $i := 1$  step 1 until  $k$  do  
           begin  $v := 0$ ;  
               for  $j := 1$  step 1 until  $n$  do  
                    $v := v + a[i,j] * x[j]$ ;  
            $y[i] := v$ ;  
           end;

The method described in the previous section would handle both algorithms equally well. The flow graph analysis described in this section will recognize a funnel in algorithm B but none in algorithm A. The funnel it recognizes in algorithm B is the block

$v := 0$ ;  
for  $j := 1$  step 1 until  $n$  do  
        $v := v + a[i,j] * x[j]$ ;

What run-time organization corresponds to a compile-time determination

of the multiplication order? Basically, the factor storage space was organized as a stack even in Chapter 4, but there the stack was never popped until all factors had been pushed. Now the factor storage space will be treated even more like a stack. Upon entering a funneling block, a marker will be placed on the stack. Upon exit of the block, factors are popped and multiplied into a  $\vec{g}$  vector (initialized to a unit vector corresponding to the funnel variable) until the marker is reached. Then the  $\vec{g}$  vector is compressed and pushed back on the stack. Upon exit of the entire algorithm, the one-row-at-a-time approach can be used to get the desired Jacobian.

## 6. IMPLEMENTATION

A compiler, called "Jake," has been written to implement the theory described in Chapters 4 and 5. It will produce subroutines for gradients or Jacobians from the text of the subroutine for the function itself. It has been designed to provide a practical tool for numerical analysts currently hesitant to use numerical methods requiring derivatives.

This chapter describes Jake: its input, its output, what it does, what to expect from it and what not to expect from it. The description is aimed at the person who wants to use and understand Jake; it is clearly not adequate for one who needs to make major changes to Jake. Jake is a multi-pass compiler and hence rather large, so a detailed description of it would unnecessarily clutter up this thesis. Fortunately, the art of writing large compilers is more and more being transformed into a real science, and the newly emerging precepts of that science have been followed in the construction of Jake wherever possible.

### 6.1. A User Description

Many of the design decisions regarding Jake were guided primarily to suit the user in the situation characterized by the following scenario:

The user is involved in a problem requiring some kind of functional iteration such as optimization of a function with respect to many variables. So far the user has avoided iteration schemes requiring knowledge of derivatives, such as Newton iteration or Fletcher & Powell iteration. Instead, the user employs an iterative scheme only requiring to sample the function, giving up the better convergence characteristics of the former methods. Then the user learns about a new method of obtaining derivatives which might tip the balance in favor of functional iteration with derivatives.

For such a user, Jake must be able to accept existing programs for the function with minimal changes. Hand-translation of programs is very error-prone and must be avoided if at all possible. Hence, the input language of Jake should be FORTRAN.

Similarly, the scenario virtually dictates that the output of Jake should be a subroutine written in FORTRAN, deviating from the ANSI 1966 standard only in trivial situations (e.g. where the input program violates the standard in the same way). Having the output of Jake appear in FORTRAN rather than in the machine code for a particular machine enormously enhances the flexibility and portability of Jake while simplifying its design. For example, it allows running Jake on a different machine from the one that will run Jake's output.

If the input and output language of Jake are virtually determined by considering the user for which it is intended, the language in which Jake itself is written is not. Here the criteria are ease of development and ease of distribution with emphasis on the first, due to the restricted scope of the project leading to this thesis. It was decided to develop Jake in the language C, running under a Unix operating system. "C" is well suited as a compiler implementation language. Unix is a very convenient and hospitable operating system. Unix is a trademark of Bell Laboratories. In anticipation of later distribution, Jake has been written in a subset of C and in a style that should allow relatively easy (hand) translation into the Ratfor language. Ratfor, like C developed at Bell Labs, is a preprocessor for FORTRAN and at least as portable as FORTRAN itself.

#### 6.1.1. The Jake Input Language: how to prepare your program

The input for Jake consists of a single FORTRAN subroutine to which a CONSTRUCT statement has been added, and which is subject to certain restrictions. First, the CONSTRUCT statement will be discussed.

#### 6.1.1.1. The CONSTRUCT Statement

Consider the following example:

```

      SUBROUTINE FUNC(X,N,Y,COEF)
      REAL X(N),Y,COEF
      CONSTRUCT D(Y)/D(X) IN GRAD(N)
      Y=COEF
      DO 10 I=N
          Y=Y*X(I)
10      CONTINUE
      RETURN
      END

```

Here Y is computed as a function of X, while N and COEF are merely additional parameters. The gradient of Y as a function of X is desired, and it is to be stored in the array GRAD, i.e.

$$\text{GRAD}(Y) = \partial Y / \partial X(I).$$

Jake learns this from the CONSTRUCT statement. (Note that an ordinary FORTRAN compiler will regard the CONSTRUCT as a comment, beginning as it does with a "C" in column 1.) Without the CONSTRUCT, Jake would not assume that X is the independent variable, nor that Y is the dependent variable. By themselves, the variable names X, Y, N have no special meaning. So the following subroutine produces the same result when submitted to Jake:

```

      SUBROUTINE FUNC(U,M,V,C)
      CONSTRUCT D(V)/D(U) IN GRAD(M)
      REAL U(M),V,C
      V=C
      DO 10 I=1,M
          V=V*U(I)
10      CONTINUE
      RETURN
      END

```

In previous chapters it was convenient to always use "x" for the independent variable, but there is no reason to burden Jake with that convention.

As another example of what can be done with the CONSTRUCT statement, consider

```

      SUBROUTINE WHAT(P,Q,PI,R)
CONSTRUCT D(Q)/D(P,R) IN S(2)
      REAL P,Q,PI,R,T
      T=SIN(P*PI/4)
      Q=COS(R)/T*R
      RETURN
      END

```

This example suggests that neither the name of the function, the name of the resulting gradient, the order of parameters in the subroutine nor even the form of the independent variables is presupposed by Jake. Based on the CONSTRUCT statement, Jake will cause to be computed:

$$S(1) = \frac{\partial Q}{\partial P} \text{ and } S(2) = \frac{\partial Q}{\partial R}.$$

PI is regarded as a constant.

The next example introduces Jacobians:

```

      SUBROUTINE WHO(U,V,N,W)
      REAL U(N), V(N), W(N)
CONSTRUCT D(W)/D(U,V) IN R(N,100)
      DO 10 I=N
          W(I)=U(I)*V(N-I+1)
10      CONTINUE
      RETURN
      END

```

This CONSTRUCT statement asks for the Jacobians  $\partial W/\partial U$  and  $\partial W/\partial V$  to be computed and stored in R as follows:

$$R(i,j) = \partial W(i)/\partial U(j) \quad \text{for } i=1,\dots,n \text{ and } j=1,\dots,n$$

$$R(i,n+j) = \partial W(i)/\partial V(j) \quad \text{for } i=1,\dots,n \text{ and } j=1,\dots,n$$

In contrast, the following produces a gradient:

```

SUBROUTINE HOW(V,N,W)
REAL V(N,N),W
CONSTRUCT D(W)/D(N) IN G(1000)
W=1.
DO 10 I=1,N
    DO 10 J=1,N
10      W=W*V(I,J)
RETURN
END

```

The layout will be as follows:

$$G(I + (J-1) * N) = \partial W / \partial V(I,J)$$

So a matrix such as  $V(N,N)$  is really treated as the one-dimensional array of  $N^2$  contiguous storage locations it represents. It is the total size of the dependent variable(s) that determines the number of rows in the Jacobian (1 for a gradient); it is the total size of the independent variables that determines the number of columns in the Jacobian. So the shape of the Jacobian is determined by dependent and independent variables; the declared dimensions of the variable receiving the result must be compatible with this.

To obtain interpretable results, the receiving variable must be a matrix for a Jacobian or a gradient (but the whole gradient must then fit in the first row), or the receiving variable may be a vector for a gradient. A vector cannot receive a Jacobian.

These restrictions only apply to the resulting variable; as indicated, independent and dependent variables may have arbitrary shapes and sizes.

Finally, consider subroutine WHY, which demonstrates the use of double precision.

```

SUBROUTINE WHY(U,V,W,N)
REAL U(N),V(N),W
CONSTRUCT D(W)/D(U) IN GRAD(N)
DOUBLE PRECISION GRAD
W=1.
DO 10 I=1,N

```

```

      W=W*U(I)+V(I)
10      CONTINUE
      RETURN
      END

```

By so indicating that the result vector GR is in double precision, one asks Jake to emit all factors in double precision to perform multiplication of factors in double precision and to extract the result in double precision. At the same time, all variables that were single precision in the original program remain single precision. Some intermediate values, such as "W\*U(I)" are computed in single precision in the original program but double precision when processed by Jake. Here is a compromise between accuracy, speed and ease of handling by Jake.

Conversely, it is possible to produce a single precision gradient in an otherwise double precision computation.

#### 6.1.1.2. Restrictions on the Input Language

Though the input language of Jake is FORTRAN, it will not handle correctly all conceivable programs in all conceivable FORTRAN dialects. The purpose of this section is to indicate the limitations Jake has with regards to the input program. Eight such restrictions will be listed, with comments and explanations where appropriate.

- a) Jake will not recognize "statement functions."
- b) Jake will not recognize variables, constants or functions of type COMPLEX.
- c) The input program must not contain any of the following subroutine names, as they are reserved for Jake:

```

SPINIT EMIT0 EMIT1 EMIT2 SPGRAD SPCOPY
DPINIT DMIT0 DMIT1 DMIT2 DPGRAD DPCOPY.

```

The output of Jake will contain calls to these subroutines. They comprise the "run-time support package" associated with Jake. The first row lists the subroutines used for a single precision

Jacobian, the second row lists those for a double precision Jacobian.

There are two rather severe restrictions that are difficult to state precisely. The easiest formulation is completely safe, but it is overly restrictive:

- d') Jake cannot handle EQUIVALENCES correctly.
- e') Jake cannot handle CALLs and function references correctly (except standard built-in functions such as SIN, DLOG).

Actually, there are many EQUIVALENCES and CALLs that are harmless and that will be processed correctly by Jake. However, Jake is not able to detect which EQUIVALENCES or CALLs are harmful, and the user must assume that burden. To perform this detection, the user will need to understand the theory of Chapter 4.

A more correct statement of the restrictions follows:

- d) Jake cannot handle correctly EQUIVALENCES that change the state space.
- e) Jake cannot handle correctly CALLs and function references that affect the values of variables in the state space in such a way as to require a nonzero factor to be emitted.

As an example of the difficulties arising with EQUIVALENCE, consider:

```
CONSTRUCT D(Y)/D(X) ...
    EQUIVALENCE (U,V)
    U=X
    Y=V
    RETURN
    END
```

Here the flow graph analysis used by Jake is not able to trace a path of nonzero factors from X to Y and it will conclude that  $\partial Y / \partial X = 0$ . However, the following EQUIVALENCE is harmless:

Here the flow graph analysis used by Jake is not able to trace a path of nonzero factors from X to Y and it will conclude that  $\partial Y / \partial X = 0$ . However, the following EQUIVALENCE is harmless:

```

      .
      .
      .
CONSTRUCT D(Y)/D(X) ...
      EQUIVALENCE (A,B)
      COMMON A
      Y = A * X
      Y = B * Y
      RETURN
      END

```

where A and B are merely parameters and hence were never part of the state space anyway.

To illustrate the situation for CALLs and functions, contrast

FUNCTION ARRMAT(U,N)	FUNCTION INDMAT(U,N)
REAL U(N)	REAL U(N)
ARRMAT=U(1)	INDMAT=1
DO 10 I=2,N	DO 10 I=2,N
IF(ARRMAT.LT.U(I))ARRMAT=U(I)	IF(U(INDMAT).LT.U(I))INDMAT=I
10 CONTINUE	10 CONTINUE
RETURN	RETURN
END	END

Using the function INDMAT within a subroutine submitted to Jake is harmless, whereas the use of ARRMAT could be harmful if any of the U(I) had a nonzero derivative with respect to the independent variable.

Jake will issue a zero factor when ARRMAT is used, without ever having seen the text of the function ARRMAT.

- f) The same restrictions apply to a READ as for a CALL.
- g) There are some restrictions involving standard (built-in) functions. The following standard functions are recognized and

The following standard functions are not recognized but handled correctly nevertheless because either arguments or result are integer:

```

      FLOAT  IFIX  INT  MAX1  MIN1  AMAX0  ISIGN
      DFLOAT IABS  MOD  MAX0  MIN0  AMINO  IDIM

```

The following standard functions are not recognized and not handled correctly:

```

      ABS    SIGN    AMAX1  AMIN1
      DABS   DSIGN   DMAX1  DMIN1

```

Standard functions accepted by some FORTRAN compilers but not in the above collection will probably not be handled correctly.

h) Jake cannot handle "out-of-bound" addressing.

Most FORTRAN dialects, in contrast to the ANSI 1966 standard, allow, e.g.

```
U = A(101)
```

where

```
COMMON A(100), B(10)
```

and the effect will be as if

```
U = B(1)
```

had appeared in the program. Jake cannot handle this correctly for the same reason that EQUIVALENCE presents problems.

#### 6.1.2. Jake Output

The output of Jake is in FORTRAN, adhering to the ANSI 1966 standard at least as much as the input does. For instance, on input it is acceptable to use REAL\*8 as a synonym for DOUBLE PRECISION; only the latter form will appear on output. On input it is acceptable to give Hollerith strings in single quotes, e.g. 'HELLO'; on output it will appear as 5HELLO.

Except for such paraphrasing of the input program, the major transformations to the input program occur at procedure entry, at relevant assignment statements, and at procedure exit. The changes at procedure entry include changes to the SUBROUTINE statement, additional declarations of variables introduced by Jake, and a number of

relevant assignment statements, and at procedure exit. The changes at procedure entry include changes to the SUBROUTINE statement, additional declarations of variables introduced by Jake, and a number of initializations.

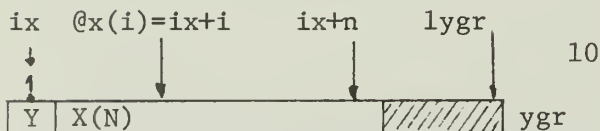
Perhaps the best way to convey an idea of what Jake does to an input program is to show some examples. The first example given has been paraphrased in order to highlight certain important features at the expense of others. The second and third examples are accurate in every detail and have not been "retouched" in any way.

Example 1:  $y = \prod_{i=1}^n x_i$ , retouched

```

      subroutine mult(x,n,y)
      real x(n),y
      construct d(y)/d(x) in gr(n)
      y=1.
      do 10 i=1,n
10         y=y*x(i)
      return
      end

```



```

SUBROUTINE MULTJ(X,N,Y,GR,YGR,LYGR)
INTEGER N,I,LYGR
REALX(N),Y,GR(N),YGR(LYGR)
IX=1
CALL SPINIT(IX+N,LYGR)
CALL EMIT0((1))
Y=1.
DO 10 I=1,N
  CALL EMIT2((1),X(I),IX+I,Y,(1))
  Y=Y*X(I)
  CONTINUE
CALL SPGRAD(YGR,LYGR,(1))
CALL SPCOPY(GR,YGR((IX+1)),N)
RETURN
END

```

10

The array containing the product of factors multiplying from the left is YGR(LYGR). SPINIT checks whether the array is large enough and initializes the factor storage space. EMIT<sub>i</sub> writes a factor to the factor storage space, where  $i = 0, 1, 2$  indicates the number of right hand side variables in the factor. SPGRAD performs the factor multiplication starting from a unit vector with a one in the position for Y. SPCOPY extracts the gradient requested from the array YGR. The resulting subroutine has YGR, LYGR as parameters because FORTRAN does not allow declaration of run-time array bounds in any other way; hence the program calling MULTJ will have to allocate space for YGR.

Example 2:  $\vec{y} = A\vec{x}$ , unretouched.

```

      subroutine matmul(a,n,x,y)
      real a(n,n), x(n), y(n), jac
construct d(y)/d(x) in jac(n,n)
      do 20 i=1,n
        w=0
        do 10 j=1,n
          w=w+a(i,j)*x(j)
10          continue
        y(i)=w
20        continue
      return
      end

```

```

      SUBROUTINE MATMUL(A, N, X, Y, JAC, YJAC, LYJAC)
      INTEGER N, I, J, LYJAC, IJAC, RJAC, LJAC, JJAC, IY, IX
      REAL A(N,N), X(N), Y(N), JAC(N,N), W, YJAC(LYJAC), TJAC
      *, TJAC1, TJAC2, TJAC3, TJAC4, TJAC5, TJAC6, TJAC7, TJAC8
      IY=10
      IX=IY+N
      CALL SPINIT(IX+N, LYJAC)
      DO 8001 I=1, N
      CALL EMIT0(1)
      W=0.
      DO 8002 J=1, N
      CALL EMIT1(IX+J, A(I, J), 2)
      TJAC=A(I, J)*X(J)
      CALL EMIT2(1, 1, 2, 1, 1)
      W=W+TJAC
8002 CONTINUE
      CALL EMIT1(1, 1, IY+I)
      Y(I)=W
8001 CONTINUE
8000 CONTINUE
      RJAC=0
      LJAC=N
      DO 8003 JJAC=1, LJAC
      CALL SPGRAD(YJAC, LYJAC, IY+JJAC, RJAC, IJAC)
      CALL SPCOPY(JAC, IJAC, N, YJAC(IX+1), N)
8003 CONTINUE
      RETURN
      END

```

In this example it can be seen how SPGRAD is called in a loop once for every row of the Jacobian. SPGRAD and SPCOPY have some arguments, suppressed in example 1, that jointly keep track of addressing in JAC: RJAC (maintaining a row count, updated in SPGRAD), IJAC (maintaining an index in JAC, updated in SPGRAD and SPCOPY), and N (indicating how far apart in memory the consecutive elements of a row of JAC are). The name of the subroutine is MATMUJ, formed from the original MATMUL by appending a "J", then dropping the penultimate character to keep the entire name within 6 characters. Many compilers allow names of 7 or more characters, but all accept 6-character names. Note also that Jake has introduced temporary variables TJAC, TJAC1,..., TJAC8, of which only the first one is actually used.

Another interesting feature exhibited by this example is that not all REAL variables partake in the state space. So the matrix  $a(n,n)$  is regarded as being outside the state space; hence no space for  $a(n,n)$  is required in the array YJAC. The determination of what goes into the state and what doesn't is made by Jake through flow graph analysis. A variable  $v$  such that either  $\partial v / \partial x = 0$  or  $\partial y / \partial v = 0$  can be proven through flow graph analysis is called "irrelevant" and accorded no place in the state space. Assignment statements updating such variables  $v$  need not emit factors either.

## Example 3:

```

      subroutine kloo(x,y,z)
      real x(3),y(2)
      real*8 z,dd
      construct d(z)/d(x,y) in dd(4,6)
      z=1. d0
      do 10 i=1,3
10         if( x(i).gt.0. ) z=z*log(x(i))
      do 20 i=1,2
          if( y(i).lt.0. ) goto 20
          z=z*y(i)
20         z=z**2
      return
      end

```

```

SUBROUTINE KLOOK(X,Y,Z,DD,YDD,LYDD)
  INTEGER I,LYDD,IDD,RDD,LDD,JDD,IY,IX
  REAL X(3),Y(2),ALOG
  DOUBLE PRECISION Z,DD(4,6),YDD(LYDD),TDD,TDD1,TDD
*2,TDD3,TDD4,TDD5,TDD6,TDD7,TDD8
  IY=10
  IX=IY+2
  CALL DPINIT(IX+3,LYDD)
  CALL DMITO(1)
  Z=1. D0
  DO 8001 I=1,3
    IF(X(I).LE.0.)GOTO 8002
    CALL DMIT1(IX+I,1,D0/X(I),2)
    TDD=ALOG(X(I))
    CALL DMIT2(1,TDD,2,Z,1)
    Z=Z*TDD
8002  CONTINUE
8001  CONTINUE
  DO 8003 I=1,2
    IF(Y(I).LT.0.)GOTO 20
    CALL DMIT2(1,Y(I)+0.D0,IY+I,Z,1)
    Z=Z*Y(I)
20    CONTINUE
    CALL DMIT1(1,Z+Z,1)
    Z=Z**2
8003  CONTINUE
8000  CONTINUE
  RDD=0
  CALL DPGRAD(YDD,LYDD,1,RDD,IDD)
  CALL DPCOPY(DD,IDD,4,YDD(IX+1),3)
  CALL DPCOPY(DD,IDD,4,YDD(IY+1),2)
  RETURN
  END

```

In this third example, note the types of all variables and expressions. In particular, all arguments to DMIT are of consistent type. It is also interesting to see how flow of control has been reworked. This is necessary because FORTRAN allows only a single statement following a "logical IF."

The generation of names and statement numbers by Jake deserves an additional comment. Except for the subroutine names used by Jake, such as "SPGRAD," no names or statement numbers used by Jake will ever interfere with names and statement numbers in the original program. So the name "IX" used by Jake in Example 3 would never have been generated if KLOO itself had contained IX. Jake will try small perturbations of "IX" until one is found that is not in KLOO.

## 6.2. How Jake Works

As mentioned earlier, the description of the innards of Jake will be very brief. Jake consists of four passes:

- 1) the lexical preprocessor
- 2) the parser
- 3) the tree building and flow analysis pass
- 4) the differentiator and output constructing pass

To run the program produced by Jake, we need

- 5) the "run-time support" package

### 6.2.1. The Lexical Preprocessor of Jake

The lexical preprocessor of Jake takes the input program and reworks it to give it a recognizable lexical structure. This is necessary because FORTRAN attaches significance to the column a character is in; FORTRAN does not reserve its keywords; FORTRAN does not attach significance to spaces between variables or in the middle of variables, or between keywords and variables; FORTRAN does not allow parsing with limited look-ahead. In short, FORTRAN has nothing like the lexical structure one takes for granted in more recent, more decent languages. The lexical preprocessor, working on an entire FORTRAN statement at a time, will:

- a) eliminate comments (but it keeps the CONSTRUCT);
- b) collect the statement fields of a statement and all (if any) continuation statements following it into a single line of arbitrary length;
- c) separate all lexemes from each other by spaces or special operator symbols;
- d) translate keywords to lower case and variables to upper case;
- e) rework statement numbers.

An example follows.

<u>input</u>	<u>after processing</u>
subroutine abc(d)	subroutine ABC(D)
construct d(d)/d(e) in f(3)	construct (D)(E)F(3)
COMMON/g/e(3)	common /G/E(3)
c this is a comment	D=E(1)+E(2)*SQRT(E(3))
d=e(1)+e(2)*sqrt(	do 10 I=1,5
&e(3))	DO20J=3
do 10 i=1,5	:10 continue
do 20 j=2	end
10 continue	
end	

The techniques used in the preprocessor are mostly ad-hoc, and not particularly interesting. The only remarkable aspect of the preprocessor is that it works, and works fast. The preprocessor should prove useful in its own right.

#### 6.2.2. Jake's Parser

In striking contrast to the preprocessor, there is nothing ad-hoc about the parser. The parser was generated by the YACC [JOH75] parser generator system running under Unix. Given a BNF description of the FORTRAN grammar, YACC produces an LALR parser for FORTRAN which will run when supplied with a lexical scanner. The preprocessor leaves the input program of Jake in a form that allows a lexical scanner and a BNF grammar for FORTRAN to be written. The grammar is simplified by the restriction that COMPLEX constants not occur and by the fact that

several statements (such as the `FORMAT` statement) need not be parsed beyond the identifying keyword. Statements, like `FORMAT`, that are simply to be carried along in `Jake` to be placed in the output without change; statements, moreover, that do not affect the outcome of flow analysis, are called "carryalongs." After reading the keyword, the lexical scanner simply stores the rest of the statement, unanalyzed, in a file for later retrieval. Hence, no BNF needs to be specified for `FORMATs`, `WRITE` statements, `DATA` statements, etc. Except for the "carryalong" feature, the lexical scanner is fairly standard. It was modeled after [COM78]. The result of parsing the (preprocessed) input program is a string of tokens in a postfix representation of the program tree. That tree is not actually built until the third pass. The tokens in the postfix representation may represent arithmetic operations such as `+`, `-`; they may represent variables (such tokens are parameterized by an index into a nametable containing the name of the variable), constants (likewise); they may represent statements, such as `COMMON` or `IF`.

#### 6.2.3. The Tree-building and Flowgraph Analysis Pass

Out of the postfix token string produced by the parser a program tree is constructed. A symbol table is built at the same time. The symbol table collects declarative information for variables: name, type, dimensions, initialization. Declarations are not incorporated in the tree. In essence, the program tree could be executed directly, at least by some kind of abstract machine. All nodes in the tree are either binary, unary, or null-ary. Each token has a fixed "arity" of 2, 1, or 0. The null-ary tokens comprise the leaves of the tree. Though the leaves have no descendants, they may carry additional information such as pointers to symbol table entries, pointers to name table entries or pointers into the carryalong file.

From the program tree the flow graph is obtained, by breaking up the tree in pieces corresponding to a single statement (an `IF` statement will get broken into two statements) and associating each such statement with a node in the flow graph. `GOTOs`, `IFs`, `DOs` and statement labels define the edges in the flow graph. Each node lists the "left hand

side" variable that is affected by the statement (if any), as well as the right hand side variables that may affect the left hand side variable. The flow graph analysis performed is itself fairly standard and straightforward. The literature on flow graph analysis is extensive. See e.g. [KAM76]. The questions that Jake attempts to answer through flow graph analysis are whether for a left hand side variable  $v$  in a certain node we can say for certain that  $\partial y / \partial v = 0$ , and if not, whether for the right hand side variables  $w$  in the same node we can say for certain that  $\partial w / \partial x = 0$ . The zero/possibly-nonzero character of  $\partial w / \partial x$  is propagated through the program much in the same way as the uninitialized/possibly-initialized character of a variable is propagated. The latter is a standard example of a characteristic determinable by flow graph analysis. The determination about  $\partial y / \partial v$  being zero is essentially similar and can be visualized as an initialization problem for the program running backwards in time.

Statements having a left hand side that does not in any way contribute to the final value of  $y$  or cannot be traced back to the values of  $x$  are called "irrelevant." Variables that are given values in "irrelevant" statements only are called irrelevant variables. Real-valued variables that are found to be irrelevant through flow graph analysis need not be considered as part of the state space. No factors need be emitted for irrelevant statements. The statement

$$v := w * p$$

even if relevant, need emit only  $\partial v / \partial w$  and may omit  $\partial v / \partial p$  if flow graph analysis can prove that  $\partial p / \partial x = 0$ . So flow graph analysis and the detection of (possible) relevance allows a reduction in the state space, the size of the row vector that needs to be supplied, the number of factors to be emitted, and even the size of the factors that are emitted. The tree, the symbol table, and relevancy information is passed on to the last pass.

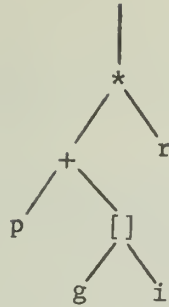
#### 6.2.4. The Differentiator and Output Building Pass

This pass has to emit the factors and must therefore be able to differentiate formulas. The approach to differentiation chosen in Jake is similar to that in Warner [WAR75], Joss [JOS76] and Kedem [KED77].

It consists of splitting up more complicated formulas in simple ones, using assignments to newly created temporary variables. This way a simple differentiation scheme suffices to create expressions that have common subexpressions of any complexity. The would-be common subexpressions of the derivatives of the original expression show up as exactly those temporary variables. The simple differentiation scheme alluded to is one where we merely need to know the derivative of each elementary mathematical operation with respect to each of its operands and nothing more. If some of these operands are flagged as "irrelevant" or "constant," so much the better. Because operations can be only binary, unary or null-ary, it suffices to have three factor emission routines, EMIT2, EMIT1, and EMIT0. (Three more are needed for double precision gradients/Jacobians.)

In addition to differentiation, the fourth pass has to be able to generate addresses @v[i] for addressing in the state space. It has to be able to generate declarations and initializations. All this essentially depends on the information in the symbol table being sufficient--and that is the responsibility of the previous pass. The fourth pass must generate program text for performing factor multiplication and extraction of the desired information, one row at a time, from the row vector into the receiving variable. Generation of this part of the output program is tremendously simplified by careful design of the procedure interfaces (e.g. parameter lists) of SPGRAD and SPCOPY.

As the differentiation of relevant assignment statements results in a modified program tree, we still need as part of the fourth pass a set of procedures that will print the tree in a form compatible with FORTRAN rules. In addition, it is preferable to have the output appear in human-readable form, avoiding names like U00001 in favor of the names appearing in the input program or at least names reminiscent of those. So most names created by Jake are derived from the name of the variable receiving the gradient/Jacobian. Creation of statement numbers by Jake is simpler and less sophisticated. Printing the program tree



means avoiding the extremes of  $p + g(i) * r$  (which is incorrect) and  $((p) + (g(i)))) + (r)$  (which is unduly hard to read). Jake actually produces

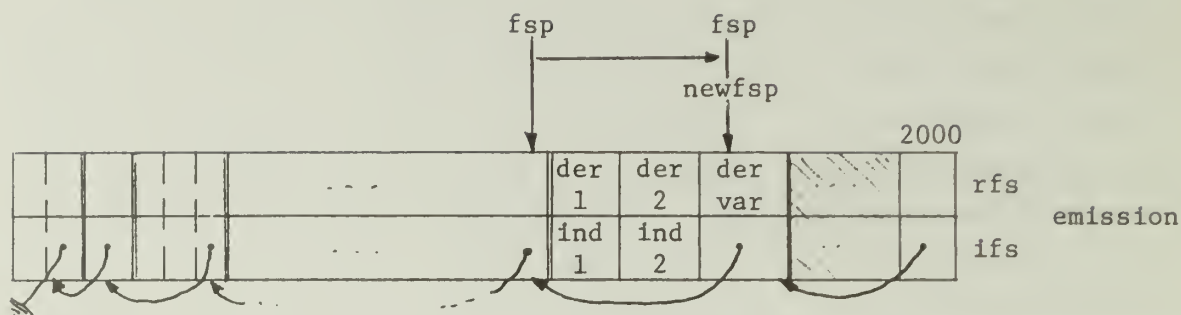
$$(p + g(i)) * r$$

but one can easily construct expressions where Jake prints redundant parentheses.

#### 6.2.5. The Run-time Support

From a glance at the example outputs from Jake the impression may have been gained that the real work is being performed in the subroutines EMIT1, SPGRAD, etc., and not in the subroutine produced by Jake. In a certain sense this is true. Barring small details involving the argument lists of SPINIT and SPGRAD, the output produced by Jake is still compatible with Joss' method. That is, the subroutines EMIT1,  $i = 0, 1, 2$ , can be written to perform right-to-left multiplication of factors in the Joss way. If it is true, then, that much of the work is done by the subroutines EMIT1, SPGRAD, etc., it does not follow that these subroutines are particularly hard to write or that they are particularly long. Moreover, these subroutines can be written once and for all; only minor changes will be required to convert from one computer to another with different disk/IO conventions. For maximal speed, subroutine SPGRAD can be written in machine code.

The text of two subroutines, EMIT2 and SPGRAD, will be presented integrally on the next two pages. The other subroutines are either too similar or too simple to merit discussion here.

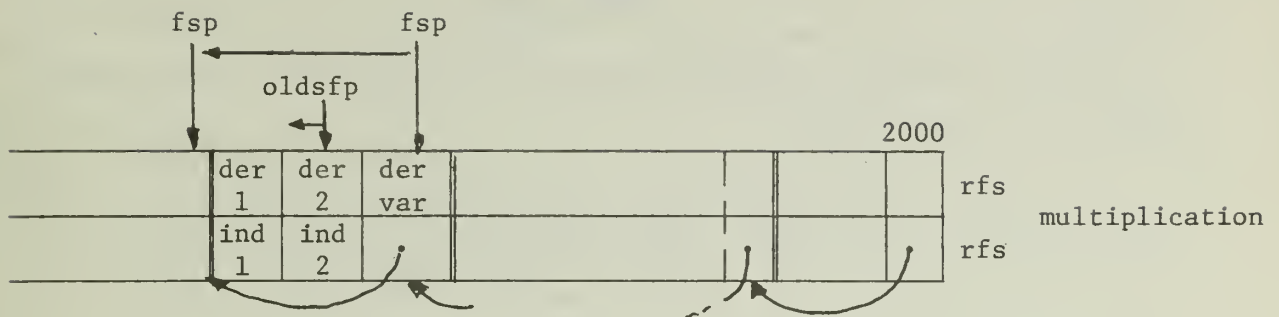


```

subroutine emit2(ind1, der1, ind2, der2, depvar)
integer ind1, ind2, depvar, fsp
real der1, der2
common/factor/rfs(2000), ifs(2000), fsp, nfs

newfsp=fsp+3
if(newfsp.lt.2000) goto 10
    ifs(2000)=fsp
    call putfac(nfs, rfs, 4000)
    fsp=0
    newfsp=3
10  ifs(fsp+1)=ind1
    rfs(fsp+1)=der1
    ifs(fsp+2)=ind2
    rfs(fsp+2)=der2
    ifs(newfsp)=fsp
    rfs(newfsp)=depvar
    fsp=newfsp
return
end

```



```

subroutine segrad(ygrad, lygrad, id, ngrad, isgrad)
integer ngrad, isgrad, fsp, oldfsp, fsend, dervar
real ygrad(lygrad), t
common/factor/rfs(2000), ifs(2000), fsend, nfsend
do 10 i=1, lygrad
10      ygrad(i)=0.
      ygrad(id)=1.
      ngrad=ngrad+1
      isgrad=ngrad
      fsp=fsend
      nfs=nfsend

20      if(fsp.ne.0) goto 30
      if(nfs.eq.0) return
      call setfac(nfs,rfs,4000)
      fsp=ifs(2000)
30      dervar=rfs(fsp)
      oldfsp=fsp-1
      fsp=ifs(fsp)
      t=ygrad(dervar)
      if( t.eq.0. ) goto 20
      ygrad(dervar)=0.
40      if(oldfsp.le.fsp) goto 20
      ind=ifs(oldfsp)
      ygrad(ind)=ygrad(ind)+t*rfs(oldfsp)
      oldfsp=oldfsp-1
      goto 40
end

```

## 7. CONCLUSIONS

In this chapter we summarize the results presented in this thesis and point to future work.

If given a subroutine  $Af(\vec{x}, y)$  representing the function  $y = f(\vec{x})$  with  $\vec{x} = (x_1, \dots, x_n)$ , the system Jake described in this thesis is able to produce a subroutine  $Af'$  representing the gradient of  $y$ ,  $\partial y / \partial \vec{x}$ .

The table below shows how  $Af'$  produced by Jake compares with numerical differencing and with Joss' method.

algorithm	time	space
$y = f(\vec{x}) : Af$	T	S
$\partial y / \partial \vec{x} : \text{num. diff.}$	$O(nT)$	S
$\partial y / \partial \vec{x} : \text{Joss}$	$O(nT)$	$O(nS)$
$\partial y / \partial \vec{x} : \text{Jake}$	$O(T)$	$O(S)$

Jake represents a significant improvement over the work of Joss. For the first time a method for symbolic differentiation of algorithms has been developed that constitutes a viable and competitive alternative to numerical differencing. With a fast and reliable method of computing gradients, optimization methods requiring gradients become more attractive. Jake may serve to revive interest in this class of methods.

In addition to gradients, Jake can produce Jacobians as well. For a Jacobian of size  $k * n$  with  $k$  substantially smaller than  $n$ , Jake is significantly faster than Joss and numerical differencing. Some ideas that may ultimately lead to faster Jacobians for  $k = n$  have been put forward in Chapter 5.

Producing gradients through differentiation by hand is exceedingly error-prone. Numerical differencing is quite robust, but a DELTA must be chosen carefully, to balance truncation error, round-off error and

the type of error discussed in Chapter 3. Symbolic differentiation is reliable. No hand-translation is involved. Symbolic differentiation is also expected to be quite accurate, though not much theory is available in support of this and some cautionary notes are sounded in Chapter 3.

### 7.1. Experience with Jake

Several tests have been run with Jake, though Jake has not yet been applied to any real-life problems.

The outputs presented in chapter 6 stem from actual runs with Jake. Many more programs representing functions with known gradients were given to Jake and the output was found to be correct in all cases. In most of these cases, correctness was judged directly from inspection of the output subroutine. In the remaining cases, the output was actually run and the results compared with the known gradients/Jacobians. These include the example algorithms of section 7.1.1 and 7.1.2. The subsections below give the results of timing tests performed on the subroutines produced by Jake. All runs were made on a PDP11/35 with software-emulated floating point arithmetic, the machine on which Jake was developed. It should be very easy to repeat the timing tests on different machines.

#### 7.1.1. Timing Tests for a Gradient

The standard deviation of a set of numbers  $(x_1, \dots, x_n)$  was chosen as an example. The standard deviation is defined by

$$y = \sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 / n}, \text{ where } \bar{x} = \sum_{i=1}^n x_i / n.$$

A subroutine "stdev" implementing this function was given to Jake. The speed of the subroutine produced by Jake was compared to the speed of "stdev" itself, with the speed of a program simulating Joss' method for obtaining the gradient, and with numerical differencing. The results for various  $n$  are shown in the table below. Each column has been scaled

independently of all others to make the "stdev" time equal to one.

gradient	n=20	n=50	n=100	n=150
stdev (reference)	1	1	1	1
numerical differencing	10.8	33.6	93.4	
Joss	10.4	34.	91.7	132.3
Jake	8.6	22.2	18.2	17.9

### 7.1.2. Timing Tests for a Jacobian

A very simple algorithm is used as a timing test for Jacobians: the algorithm `matmul` which computes  $\vec{y} = A\vec{x}$  where  $A$  is an  $n * n$  matrix with constant elements. All methods do indeed reproduce  $A$  as their answer  $\partial\vec{y}/\partial\vec{x}$ .

	n=10	n=20
matmul (reference)	1	1
numerical differencing	3.3	14.4
Jake	8.6	30.8

### 7.2. Future Work

Several things could be done to make Jake a better tool; some small, some larger, and most of them fairly obvious.

#### 7.2.1. The += Operator

Assignments of the form  $u := u + a * v$ ; are extremely common. According to the theory of Chapter 4 they correspond to factors of the form

$$\frac{\partial u}{\partial u} \begin{bmatrix} 1. & a \end{bmatrix}$$

When multiplying this factor to the  $\vec{g}$  vector, we get in effect:

```

t := g[∂ u];
if t ≠ 0 then   g[∂ u] := 0;
                g[∂ u] := g[∂ u] + t * 1;
                g[∂ v] := g[∂ v] + t * a;
```

This could be written more efficiently as:

```
t := g[∂ u];
if t ≠ 0 then g[∂ v] := g[∂ v] + t * a;
```

We can think of the latter as the factor multiplication operation corresponding to a new type of factor, the factor emitted as a response to a "+:=" operator (cf. Algol 68) in contrast to the usual operation ":=". So "u := u + a \* v" is regarded as "u +:= a \* v" and the factor emitted is

$$\begin{array}{c} + \\ \backslash \partial v \\ \partial u \boxed{a} \end{array}$$

Incorporating this change into Jake would save many multiplications of a number by one. Of course, such a change will not affect the  $O(T)$  time bound as such, but it may decrease the value of the coefficient of  $T$  in  $O(T)$ . For example, for the algorithm

```
for i := 1 step 1 until n do
  y := y + x[i];
```

the number of multiplications in the gradient would decrease from  $2n$  to  $n$ .

### 7.2.2. Longer Factors

Chapter 6 shows that the factor emitted for

```
u := u + a * v;
```

is not really

$$\begin{array}{c} \backslash \partial u \quad \partial v \\ \partial u \boxed{1.} \boxed{a} \end{array}$$

as the previous subsection suggests. Instead,  $u := u + a * v$  is transformed into

```
temp := a * v;
u := u + temp;
```

with the corresponding factors

$$\begin{array}{c} \backslash \partial v \\ \partial temp \boxed{a} \end{array} ; \begin{array}{c} \backslash \partial u \quad \partial temp \\ \partial u \boxed{1.} \boxed{1.} \end{array}$$

Here another multiplication by one is introduced. It would not be trivial to change the differentiation scheme to allow it to deal directly with larger chunks of the expression at the right hand side, but it might eliminate local inefficiencies like the one shown. Not only a multiplication by one is at stake, but also the overhead associated with an additional procedure invocation.

### 7.2.3. Subroutine Calls

It would be desirable to extend Jake to enable it to cope with arbitrary calls to arbitrary subroutines as long as the text of these subroutines is also supplied to Jake.

The problems associated with such an extension are varied, but all seem technical rather than theoretical. COMMON blocks would become important, posing problems much the same as EQUIVALENCE statements do in the single subroutine case. Allowing recursive procedures would be still more difficult.

### 7.2.4. Language Extensions

Jake could be extended to recognize complex variables. It could be extended to handle a larger set of library functions, including functions not in the FORTRAN standard, such as  $\tan(x)$  and Bessel functions. The CONSTRUCT could be extended to allow the resulting Jacobian to be stored in sparse form according to some user-supplied store function; the syntax might be something like this:

```
CONSTRUCT D(Y)/D(X) USING STORE(ROW,COL,VALUE)
```

### 7.2.5. Round-off Behavior

A better understanding of accumulation of round-off in symbolic differentiation is desirable.

### 7.2.6. Faster Jacobians

Chapter 5 has already dealt with various ways in which the construction of Jacobians might be speeded up. However, more work needs to be done before symbolic construction of Jacobians will be faster than

numerical differencing by an order of magnitude.

### 7.3. Summary

Jake provides a useful, flexible and efficient tool for algorithmic differentiation. It produces gradients that can be evaluated much faster than those produced by previous methods.

A better tool can make a difference quantitatively, by allowing people to do more conveniently and more cheaply what they were doing already. A better tool can also make a difference qualitatively, by affecting certain trade-offs. It is hoped that Jake may help shift the balance in functional iteration methods in favor of those that make use of partial derivatives.

## REFERENCES

- [BEL57] Bellman, Richard E., "Dynamic Programming", Princeton Univ. Press, 1957.
- [COM78] Comer, D., "MOUSE4: An improved implementation of the RATFOR preprocessor", Software-Practice and Experience, Vol. 8, 1978.
- [DIJ76] Dijkstra, Edsger W., "A Discipline of Programming", Prentice-Hall, 1976.
- [JOH75] Johnson, S.C., "YACC - Yet Another Compiler-Compiler", C.S. Tech. Report 32, Bell Laboratories, July 1975.
- [JOS76] Joss, Johan, "Algorithmisches Differenzieren", Ph.D. Thesis, ETH, Zurich, Switzerland, 1976.
- [KAM76] Kam, J.B., Ullman, J., "Global Data Flow Analysis and Iterative Algorithms", JACM, Vol. 23, No. 1, Jan. 1976.
- [KED77] Kedem, Gershon, "Automatic Differentiation of Computer Programs", Proc. 1977 Army Numerical Analysis and Computer Conf., Madison, Wisc. 1977.
- [KUC78] Kuck, David J., "The Structure of Computers and Computations", Vol. 1, John Wiley & Sons, 1978.
- [STA76] Standish, T., Harriman, D., Kibler, D. and Neighbors, J., "The Irvine Program Transformation Catalogue", C.S. Dept., U.C. Irvine, Irvine, Cal., Jan. 1976.
- [STR69] Strassen, Volker, "Gaussian Elimination is not optimal", Numer. Math. 13, 1969.
- [WAR75] Warner, D.D., "A Partial Derivative Generator", C.S. Tech. Report 28, Bell Laboratories, April 1975.

## VITA

Born and raised in the Netherlands, Bert Speelpenning received his Engineer's Degree in Applied Mathematics from the University of Technology at Delft in 1974. His thesis project involved the design and implementation of a system for the fast computer solution of structural analysis problems according to the finite element method.

After emigrating to the United States he joined the Ph.D. program in Computer Science at Illinois in 1975. His major professional interests are language processing and the design of large software systems.



U. S. ATOMIC ENERGY COMMISSION  
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR  
DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

( See Instructions on Reverse Side )

1. AEC REPORT NO.

C00-2383-0063

2. TITLE

COMPILING FAST PARTIAL DERIVATIVES  
OF FUNCTIONS GIVEN BY ALGORITHMS

3. TYPE OF DOCUMENT (Check one):

- ☒ a. Scientific and technical report  
☐ b. Conference paper not to be published in a journal:

Title of conference \_\_\_\_\_

Date of conference \_\_\_\_\_

Exact location of conference \_\_\_\_\_

Sponsoring organization \_\_\_\_\_

- ☐ c. Other (Specify) \_\_\_\_\_

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- ☒ a. AEC's normal announcement and distribution procedures may be followed.  
☐ b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.  
☐ c. Make no announcement or distribution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

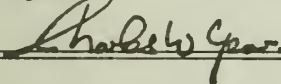
6. SUBMITTED BY: NAME AND POSITION (Please print or type)

C. W. Gear  
Professor and Principal Investigator

Organization

Department of Computer Science  
University of Illinois U-C  
Urbana, IL 61801

Signature



Date

January 1980

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

8. PATENT CLEARANCE:

- ☐ a. AEC patent clearance has been granted by responsible AEC patent group.  
☐ b. Report has been sent to responsible AEC patent group for clearance.  
☐ c. Patent clearance not required.



<b>BIBLIOGRAPHIC DATA SHEET</b>	1. Report No. UIUCDCS-R-80-1002	2.	3. Recipient's Accession No.
	4. Title and Subtitle  COMPILING FAST PARTIAL DERIVATIVES OF FUNCTIONS GIVEN BY ALGORITHMS		5. Report Date January 1980
7. Author(s) Bert Speelpenning		8. Performing Organization Rept. No. UIUCDCS-R-80-1002	
9. Performing Organization Name and Address Department of Computer Science University of Illinois U-C Urbana, IL 61801		10. Project/Task/Work Unit No.	11. Contract/Grant No. US ENERGY/ EY-76-S-02-2383
12. Sponsoring Organization Name and Address US Department of Energy Washington, DC		13. Type of Report & Period Covered Ph.D. Thesis	14.
15. Supplementary Notes			
16. Abstracts  If the gradient of the function $y = f(x_1, \dots, x_n)$ is desired where $f$ is given by an algorithm $A_f(x, n, y)$ , most numerical analysts will use numerical differencing. This is a sampling scheme that approximates derivatives by the slope of secants in closely spaced points. Symbolic methods that make full use of the program text of $A_f$ should be able to come up with a better way to evaluate the gradient of $f$ . The system "Jake" described in this thesis produces gradients significantly faster than numerical differencing. Jake can handle algorithms $A_f$ with arbitrary flow of control. Measurements performed on one particular machine suggest that Jake is faster than numerical differencing for $n > 8$ . Somewhat weaker results have been obtained for the problem of computing Jacobians of arbitrary shape.			
17. Key Words and Document Analysis. 17a. Descriptors  partial derivatives Jacobians program differentiation			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement  unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 82	
	20. Security Class (This Page) UNCLASSIFIED	22. Price	





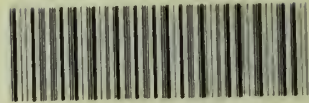








UNIVERSITY OF ILLINOIS-URBANA



3 0112 001342416